SERIES EDITOR **Macintosh Inside Out** SCOTT KNASTER

Volume I

Programmer's Guide to MPW®

Exploring the Macintosh® Programmer's Workshop

MARK ANDREWS

Programmer's Guide to MPW®, Volume I

Programmer's Guide to MPW®, Volume I

Exploring the Macintosh® Programmer's Workshop

Mark Andrews

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan

Library of Congress Cataloging-in-Publication Data

Andrews, Mark.
Programmer's guide to MPW / Mark Andrews.
p. cm. — (Macintosh inside out)
Includes bibliographical references and index.
Contents: v. 1. Exploring the Macintosh programmer's workshop
ISBN 0-201-57011-4 (v. 1)
1. Macintosh (Computer)—Programming. 2. MPW (Computer system). I. Title. II. Series.
QA76.8.M3A59 1990
005.265—dc20 90-48237
CIP

Sponsoring Editor: Carole McClendon
Technical Reviewer: Nick Pilch
Cover Design: Ronn Campisi
Text Design: Copenhaver Cumpston
Set in 10.5 point Palatino by ST Associates, Inc.

ISBN 0-201-57011-4

ABCDEFGHIJ-MW-91
First printing, December 1990

This book is dedicated to
Bhagawan Nityananda

Contents

Preface

▶ Welcome to MPW

There are many reasons to know how to use the Macintosh Programmer's Workshop. But the most important reason I know is that MPW is the most powerful programming platform that has ever been created for the Macintosh. Apple created it, Apple maintains it, and Apple's own programmers use it to develop system-level software for the Macintosh. If you own a Macintosh and a current system disk, much of the system software that makes your Macintosh run was written using MPW. And Apple is creating more and more system software using MPW every day.

MPW was introduced in 1986, and its latest revision—Version 3.2—is designed to support all the capabilities of System Software Version 7.0. But MPW still works with every Macintosh with more than 64K of ROM, as well as with every new machine that rolls off the assembly line. And that's no accident. If you program with MPW and follow Apple's human interface guidelines, Apple guarantees that you'll never be stuck with an outdated piece of software—or an outdated development system. For these reasons, and many more that you will learn about as you read this book, MPW has become the standard programming platform for the professional software developer.

Of course it is possible to write Macintosh programs with small standalone development systems; in fact, many fine Macintosh applications have been created using THINK C, Turbo Pascal, and other standalone compilers. But the Macintosh Programmer's Workshop is much more than a compiler or an assembler; it's a complete, integrated software development system and a powerful scripting language, and has a Macintosh-style interface equipped with windows, pull-down menus, and click-and-close dialog boxes.

The MPW script language has more than 120 commands that you can use to write, compile, link, and execute programs. If you need more commands, you can write your own—and that's just one way that you can customize the MPW environment.

You can add menus and menu items to the MPW menu structure. You can design your own MPW tools and execute them from scripts or custom-designed dialog boxes. You can write custom-tailored startup scripts to create a programming environment that suits your own needs and preferences. You can write other scripts that will compile and link programs in exactly the way that you want them linked and compiled. You can even use MPW to write computer languages, which will, of course, run under MPW.

MPW has so much power, and so many special features, that it would take a book to describe them all. And that's exactly why this book was written. Welcome to MPW.

▶ About This Book

Programmer's Guide to MPW, Volume I is a tutorial and reference guide for people who want to learn how to design and develop programs for the Apple Macintosh using the Macintosh Programmer's Workshop.

This book presents information about MPW in a carefully graded fashion, starting with the fundamental principles of each topic discussed and progressing to more advanced examples, with plenty of sample code and hands-on practice presented at each level of instruction.

It thus demystifies the *Macintosh Programmer's Workshop 3.0 Reference,* the giant two-volume technical manual that so many people read, but so few understand. In this book, the age-old secrets buried in the pages of the *Macintosh Programmer's Workshop 3.0 Reference* are at last brought forth into the light and explained in language that the MPW programmers of tomorrow will finally be able to understand.

Programmer's Guide to MPW, Volume I is divided into two parts. Part I covers the the MPW Editor, the MPW command language, the writing of MPW commands and scripts, the MPW menu structure (including the creation of customized menus), and MPW dialogs—including Commando dialogs. Part II focuses on advanced programming techniques, and on the relationships between MPW and the Macintosh

Event Manager, the Resource Manager, and Memory Manager. It includes source code examples that show with crystal clarity how to write an application, an MPW tool, and a desk accessory.

▶ Chapter Outline

Part I, "The MPW Shell," has four chapters:

- **Chapter 1** presents the history of MPW and describes how to install and start MPW. It introduces the Macintosh Toolbox, the Macintosh operating system, and the Macintosh Programmer's Workshop. It also explains how to use the Macintosh Toolbox and the Macintosh operating system in MPW programs.
- **Chapter 2** explores the most important features of the MPW programming environment, and explains—with the help of many hands-on programming examples—how to write and execute commands and scripts in the MPW command language.
- **Chapter 3** examines the MPW menu structure; tells how to use and customize the MPW menu bar; shows how to use dialogs and alert dialogs in MPW scripts; and introduces Commando dialogs, which can be used to execute commands by selecting dialog items rather than by typing and entering command lines.
- **Chapter 4** examines the many features of the MPW special character set, including its powerful search-and-replace and pattern-matching capabilities, and introduces more MPW commands.

Part Two, "Writing an Application," also contains four chapters:

- **Chapters 5**, **6**, and **7** explain and illustrate how to write MPW applications. They also tell how MPW interfaces with the Event Manager, the Resource Manager, and the Memory Manager.
- **Chapter 8** brings it all together and shows you how to write, compile, and link a commercial-quality application program.
- **Appendices.** The book also contains five appendices. Appendix A presents the entire MPW command set, including syntax and options. Appendix B presents the commands arranged by category. Appendices C, D, and E contain the code for the Creation Program, the application created in Part 2.

Who Needs It

Specifically, people who might have a need for *Programmer's Guide to MPW* include

- Macintosh programmers who have been using other development tools (such as THINK C or THINK Pascal) and want to learn to use MPW.
- Programmers of non-Macintosh computers who want to learn to write Macintosh programs using MPW.
- Beginning programmers who want to learn a programming language using the MPW environment (readers in this category will have to supplement the material provided in this book with texts that deal specifically with Pascal, C, assembly language, C++, or MacApp; a number of such supplementary books are listed in the "Recommended Reading" section of this Preface and the Bibliography).
- Programmers who have been using MPW, but who want to become more familiar with the subtleties of the MPW development environment (MPW is such a complex system that most MPW users do not fully understand all of its features and thus fall into this category).

More About MPW

The Macintosh Programmer's Workshop is a set of professional software development tools created for Macintosh programmers by Apple Computer, Inc. MPW is by far the largest and most feature-packed software development system for the Macintosh. Since it is an official Apple product, it is guaranteed to be upgraded as necessary in order to remain compatible with future models of the Macintosh. The current edition of MPW, Version 3.2, contains

- The Macintosh Programmer's Workshop shell, the heart of the MPW programming environment. The MPW shell includes a full-featured window-based text editor, a command-line interpreter similar to the one used in UNIX, and a dialog interface that enables the user to communicate with MPW via dialogs rather than using command lines. The shell also includes a command language that supports scripts, shell variables, control constructs, and text-editing commands.

- A linker that can combine object-code files into executable programs. The MPW linker can be accessed via menus or dialogs, or with the MPW command language. It can generate standalone applications, desk accessories, device drivers, and other varieties of programs. It can even merge code segments written in more than one language into a single application.
- Projector, a project organizer that can maintain a revision history of any project being developed under MPW. This utility can provide you with immediate access to the most recent version of any project under development, while saving backup files that can be used to re-create any earlier version. Projector is thus both an archiving tool and a backup utility. It can be used either by a single programmer or by a large, networked development team.
- MacsBug, Apple's assembly language debugger for the Macintosh.
- A set of tools that can measure the performance of programs.

In all, MPW provides more than 120 built-in tools and scripts for program developers. All of its tools are supported with a comprehensive online help command.

▶ Apple Products for the MPW User

In addition to the materials that come with the Macintosh Programmer's Workshop, many products designed to be used with MPW are available separately, both from Apple and from other suppliers. Additional MPW products from Apple include

- The Macintosh Programmer's Workshop Assembler, a tool for writing 680X0 assembly language programs.
- Macintosh Programmer's Workshop Object Pascal, a package for developing programs in Pascal.
- Macintosh Programmer's Workshop C, a C compiler that provides everything you need to develop Macintosh programs in C.
- Macintosh Programmer's Workshop C++, a C++ translator for developing object-oriented C++ programs. MPW C++ is a precompiler package designed to be used with MPW C.
- MacApp, a set of object-oriented libraries designed to speed up and simplify the process of developing Macintosh software in either Object Pascal or C++. To use MacApp, you must also have either MPW Object Pascal or MPW C++.

- The Symbolic Application Debugging Environment (SADE), a source-level debugger for programmers using C, Pascal, C++, and MacApp.
- A resource editor called ResEdit, plus other tools for creating and managing resources.
- The MPW IIGS Cross-Development System, a kit for developers who want to use the Macintosh and the MPW programming environment as a cross-development platform for writing Apple IIGS software. The MPW IIGS Cross-Development System contains modules that can be used to develop Apple IIGS software in C, Pascal, or assembly language.

▶ Other MPW-Based Products

MPW-based products from sources other than Apple include

- The AdaVantage MacProfessional Developer Kit, a development system for Ada programmers from Meridian Systems, Inc.
- Aztec MPW C, from Manx Software Systems.
- Language Systems FORTRAN, from Language Systems.
- MacFortran/MPW, from Absoft Corporation.
- TML Pascal II, from TML Systems.
- The TML Source Code Library II, a large collection of advanced programming examples written in TML Pascal, from TML Systems.
- Oracle for Macintosh, a powerful relational database from Oracle Corporation.

▶ Where to Buy MPW Products

The Macintosh Programmer's Workshop and all of the MPW-related products just mentioned can be obtained from Apple software dealers or from APDA, the Apple Programmer's and Developer's Association. APDA's address is

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mail Stop 33G
Cupertino, California 95014-6299

What You'll Need to Know to Use This Book

Programmer's Guide to MPW, Volume I: Exploring the Macintosh Programmer's Workshop is a complete tutorial, so you won't have to be an MPW wizard to understand it. If you do know MPW, so much the better; the Macintosh Programmer's Workshop is such a feature-packed programming platform that there is practically no one out there who knows as much about MPW as there is to know.

To understand the programming examples presented in this book, you'll have to have at least some knowledge of C , Pascal, or—preferably—both. If you're a beginning programmer and want to learn MPW and a programming language at the same time, you have undertaken quite a challenge. But with this book, a lot of persistence, and a good basic text on the language you want to learn, it can be done.

If you are not an assembly language programmer, it would also be a good idea to learn at least some 680X0 assembly language. All Macintosh compilers produce machine-language code and the MacsBug debugger presents you with a screenful of machine code when it detects an error. However, you don't have to know assembly language to use the source-level debuggers that are available for the Macintosh, such as the Symbolic Application Debugging Environment (SADE).

Recommended Reading

Programmer's Guide to MPW is designed to supplement—not to replace—the technical reference manuals that are supplied with the MPW system and its various compilers. You should study this book along with the documentation that came with your Macintosh and your MPW system.

Another book you should definitely own is *Inside Macintosh,* the definitive reference work for the Macintosh programmer. *Inside Macintosh,* packed into six hefty volumes, contains detailed instructions for making every call in the Macintosh Toolbox and operating system. It also provides a wealth of useful information about the Macintosh, its system software, and its hardware architecture.

Inside Macintosh is part of the Apple Technical Library, a body of work by Apple and published by Addison-Wesley. Two other useful books in the Technical Library series are *Technical Introduction to the Macintosh Family* and *Programmer's Introduction to the Macintosh Family,* which is now available in its second edition.

If you're interested in developing commercial software for the Macintosh, you should also be familiar with still another volume in the Technical Library, *Human Interface Guidelines: The Apple Desktop Interface.*

If you need detailed technical information on specialized topics, the Macintosh Technical Library offers such titles as *Macintosh Family Hardware Reference,* the *Apple Numerics Manual,* and *Designing Cards and Drivers for the Macintosh II and Macintosh SE.*

Many other books that can help you in your quest to learn MPW and the Macintosh are listed in the Bibliography.

Acknowledgments

It takes more than one person to write a computer book; it takes an army. And the troops who worked on this book are the finest I've seen anywhere.

Thanks to Carole McClendon of Addison-Wesley, who asked me to write the book and whose confidence never flagged, even during the rough spots; Series Editor Scott Knaster, whose knowledge of the Macintosh is awesome (he can tell you more about the Macintosh off the top of his head than most people can find in the manuals); Project Editor Joanne Clapp Fullagar, whose eagle eye and attention to detail kept the pages pristine; Production Supervisor Diane Freed, who made sure that everything stayed on track production-wise; and Associate Editor Rachel Guichard, who provided valuable reference materials and kept the copy flowing.

Special thanks to Copy Editor Margaret Hill and Technical Reviewer Nick Pilch, who spent many long hours over hot copy to make sure that no errors crept in.

Also: Jordan Mattson, Peter Norton, Peter Alley, and Linda Suits at Apple, for technical support; Kirk Chase, who published an excerpt of Chapter 4 in *MacTutor* magazine; and all of the support people at Addison-Wesley and Apple, without whose help this book would have never been possible.

PART ONE

▸ The MPW Shell

In Part One, we will take a closeup look at each major part of the Macintosh Programmer's Workshop (MPW) programming environment, or MPW shell. These parts include

- worksheet window
- command language
- scripts
- menu structure
- dialogs (including Commando dialogs)
- editor

This part has four chapters:

- ▸ Chapter 1 examines the most important features of the Macintosh, tells how the Macintosh User Interface differs from the user interfaces used by other computers, and introduces the Macintosh Toolbox, the Macintosh operating system, and the Macintosh Programmer's Workshop. It also explains how to use the Macintosh Toolbox and the Macintosh operating system in MPW programs.
- ▸ Chapter 2 explores the most important features of the MPW programming environment, and it explains—with the help of many hands-on programming examples—how to write and execute commands and scripts in the MPW command language.

- Chapter 3 examines the MPW menu structure; tells how to use and customize the MPW menu bar; shows how to use dialogs and alert dialogs in MPW scripts; and introduces Commando dialogs, which can be used to execute commands by selecting dialog items rather than typing and entering command lines.
- Chapter 4 describes the many special characters used in the MPW command language and provides many tables and examples showing how they are used.

1 ▶ MPW and the Macintosh

This book is about a computer called the Macintosh and a software development system called the Macintosh Programmer's Workshop: two products that are, quite literally, made for each other.

The Macintosh Programmer's Workshop was designed for the Macintosh—and today's Macintosh was designed using MPW.

The Macintosh has come a long way since it was introduced to the public in 1984. The first Macintosh—a real Model T, by today's standards —had a tiny 9-inch black-and-white screen, only 128K of RAM, and a single-sided, 400K floppy-disk drive. Today's top-of-the-line models have giant color screens, can operate at speeds of up to 40MHz, and can address up to 1 gigabyte of memory.

Macintosh engineers took one of their boldest steps forward to date with the announcement of System Software Version 7.0, which supports the interactive editing of data across applications running simultaneously on the same computer and even operating simultaneously across a net. Among its many other new features, System 7 has a new multitasking Finder that's integrated into the operating system; includes outline fonts, which can be expanded to any size without jagged edges; and allows the Macintosh user to access databases running on remote systems.

Along with System 7, Apple introduced Version 3.2 of its official Macintosh software development system, the Macintosh Programmer's Workshop. MPW 3.2 includes new object-code libraries that support all of the new features of System 7—and has a host of new features of its own. It has a new editing window that can be split into as many as 20 scrollable panes; a new Browser that can find premarked sections in any text document; and StreamEdit, a new non-interactive, script-driven text editor that can reformat a document at the touch of a button.

MPW 3.2 also has new and faster tools for linking programs and creating object-code libraries, new features that support development for the 68040 microprocessor, and even new and improved C and Pascal compilers.

With the introduction of System 7 and MPW 3.2, Apple has again placed the Macintosh at the leading edge of microcomputer design. And this book can help you program today's Macintosh using today's Macintosh software development system.

In this chapter, you'll see how the Macintosh and the Macintosh Programmer's Workshop grew up together, and how they work together now.

▶ The Macintosh Story

Henry Ford didn't invent the automobile, and Apple didn't invent the mouse, windowed displays, or pull-down menus. But the Ford Model T marked the beginning of the age of the automobile, and the Apple Macintosh may one day be remembered as the computer that redefined the relationship between the machine and humanity.

Apple unveiled the Macintosh on January 24, 1984, at a gala celebration at the company's headquarters in Cupertino, California. At a multimedia presentation that featured a talking Macintosh and ended with a standing ovation, Apple co-founder Steven Jobs hailed the new machine as "the computer for the rest of us" and predicted that it would usher in a new era in the history of the computer industry.

It took a while for Jobs' prediction to come true, but we all know that it finally did. Due to some serious design limitations and an almost total lack of supporting software, the Macintosh got off to a shaky start in the marketplace. However, computer users soon began falling in love with windows, pull-down menus, and the mouse—and Apple, which had begun its life in a garage in Cupertino, had spread its wings, and was growing into a multi-billion-dollar corporation.

As revolutionary as it was—and as willing as Jobs was to take credit for it—the Macintosh was not the first commercially available computer to be built around an interface that featured movable windows, pull-down menus, icons, and a mouse. That distinction belonged to the Lisa, an Apple product that was introduced a full year ahead of the Macintosh, but never took off in the marketplace and was discontinued in 1985.

By the Way ▶

The Macintosh Grows Up. The first Macintosh was equipped with only 64K of ROM and 128K of RAM, and had a single-sided 400K floppy-disk drive and a keyboard with neither keypad nor arrow keys. A little later Apple introduced the Macintosh 512K, which had 512K of RAM but was otherwise just like the original model.

The first Mac that could be called a major upgrade was the Macintosh Plus, which had 1 megabyte of RAM, 128K of ROM, and a double-sided, 800K floppy-disk drive. Then came the Macintosh 512K Enhanced, which had 512K of RAM, 128K of ROM, and an 800K disk drive.

Since the introduction of the original Macintosh in 1984, the Macintosh family has expanded to include such illustrious members as the Macintosh IIfx, which features a 40MHz 68030 microprocessor, a 68882 floating-point coprocessor, and a built-in 32K static RAM (SRAM) cache that stores the processor's most frequently used instructions to increase its processing speed.

Other standard features of the IIfx include 4 MB of RAM; a dedicated SCSI DMA (Small Computer Systems Interface/Direct Memory Access) channel, which reduces the workload of the main processor; and dedicated I/O processors, which increase system efficiency. The Macintosh IIfx has six NuBus expansion slots that can accommodate multiple video, communications, networking, and other expansion cards, and can be purchased with a built-in 80 MB or 160 MB disk drive.

▶ The Mouse, the Lisa, and the Macintosh

The mouse, the peripheral that made the Macintosh User Interface possible, was developed in the 1960s at the Stanford Research Center in Palo Alto. Over the next decade, at its Palo Alto Research Center (PARC), Xerox developed a series of experimental workstations that coupled the mouse with bit-mapped screens, windows, icons, and pull-down menus.

In 1979, after being persuaded to invest $1 million in Apple stock, Xerox invited Jobs and a team of Apple engineers to tour its PARC research facility and take a look at the work going on there. What Jobs and his party saw during the field trip was a computer far different from anything they had ever encountered. Instead of a standard 80-column, 24-line text screen, it had a high-resolution text-and-graphics screen with windows and icons that could be manipulated with a mouse. Instead of standard multilevel menus that users had to thread their way through, it had a menu bar from which any item could be selected, at any time, with a click and drag of the mouse.

Jobs and his team were so excited by the novelty of all of this that their enthusiasm became contagious. Jobs hired several other engineers away from Xerox and put them to work developing new Apple designs.

The initial result of their efforts was the Apple Lisa, a $10,000 computer with an even more elegant interface than the one that Xerox had demonstrated. The Lisa, introduced in 1983, had a bit-mapped black-and-white screen that emulated a desktop, with pictorial icons representing folders and the documents they contained. To open a folder or launch an application, the user merely clicked on the appropriate icon with the mouse. Files and folders appeared inside movable, resizable windows, and all of the options available to the user could be accessed at any time via pull-down menus.

Furthermore, when you typed or drew in an application window, your work was displayed on the screen in true WYSIWYG ("what you see is what you get") fashion; italic text appeared on the screen in italics, bold type appeared as bold, and the shapes and sizes of text and graphics were the same as they would be if they were printed out on paper. So, when you printed out a document that you had prepared on a Lisa, everything looked the same on paper as it had looked on the screen.

▶ Lisa Bites the Dust

The Lisa, as impressive as its innovations were, did not turn out to be a smashing success. It was not compatible with any other computer on the market—not even with earlier Apples—and its price was just too high. Although it won critical acclaim, few customers were willing to pay $10,000 for such a pretty new toy. In 1985, after the Macintosh had been introduced and had been on the market for a little over a year, the Lisa was finally discontinued.

Fortunately the Macintosh—a computer designed to offer many of the Lisa's features, but at much less cost—was on the market and getting up some real steam by the time the Lisa died. Since the original

Macintosh made its debut in 1984, Apple has introduced an average of two new and improved models each year, and the evolution of the Mac shows no signs of slowing down.

Table 1-1 traces the evolution of the Macintosh, listing the members of the Macintosh family tree and some of their most important specifications.

Table 1-1. Specifications of Macintosh computers

Model	*CPU*	*Memory*	*Input Devices*	*Internal Storage*
Original Macintosh	8 MHz 68000	128K	Macintosh Keyboard Macintosh Mouse	400K disk drive
Macintosh 512K	8MHz 68000	512K	Macintosh Keyboard Macintosh Mouse	400K disk drive
Macintosh Plus	8 MHz 68000	1 MB	Mac Plus Mouse Mac Plus Keyboard Apple Scanner	800K disk drive
Macintosh 512K Enhanced	8 MHz 68000	512K	Macintosh Keyboard Macintosh Mouse	800K disk drive
Macintosh SE	8 MHz 68000	1 MB	Macintosh Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 20SC,40SC
Macintosh SE/30	16 MHz 68030/68882	1 MB	ADB Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 40SC,80SC
Macintosh Portable	16 MHz 68000	1 MB	Apple Desktop Mouse Apple Trackball* Apple Keyboard* Apple Scanner Apple Extended Keyboard Numeric Keypad	Apple FDH SuperDrive; Portable Internal 40SC Hard Disk Port
Macintosh IIcx	16 MHz 68030/ 68882	1 MB 4 MB	ADB Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 40SC,80SC

Table 1-1. Specifications of Macintosh Computers (continued)

Model	*CPU*	*Memory*	*Input Devices*	*Internal Storage*
Macintosh IIx	16 MHz 68030/68882	1 MB 4 MB	ADB Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 40SC, 80SC, 160SC
Macintosh IIci	25 MHz 68030/68882	1 MB 4 MB	ADB Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 40SC, 80SC
Macintosh IIfx	40 MHz 68030/68882	4 MB	ADB Mouse Apple Keyboard Apple Extended Keyboard Apple Scanner	Apple FDHD SuperDrive; Internal Hard Disk 80SC, 160SC

(*) Built in

▶ The MPW Story

When Apple introduced the original Macintosh in 1984, program developers rushed out to buy it—and then found out that there wasn't much they could do with it as far as program development was concerned. The source code for the Macintosh had been written not on a Macintosh, but on a Lisa—using a Pascal compiler and a 68000 assembler—and when you looked at the specifications of the original Macintosh, it was easy to understand why. The original Macintosh had only 128K of RAM and a single 400K floppy-disk drive, and those limitations made serious program development on a Macintosh all but impossible. Apple, recognizing the futility of trying to write Macintosh programs on a Macintosh, offered software manufacturers a development package that included a Lisa and a set of cross-development tools that could be used to develop programs for the Mac.

As the Macintosh evolved into a more powerful computer, and its memory and storage capabilities increased, Apple began to recognize the need for a program development system that would run on a Macintosh platform. The first product aimed at filling that need was the Macintosh 68000 Development System, or MDS. MDS provided programmers with a machine-language assembler and some support tools, but it did not include a compiler for developing programs in

Pascal, C, or any other higher level language. BASIC and Pascal packages from third-party manufacturers soon began showing up in the software marketplace, however, and Apple then decided that it was about time to start working on a full-scale Macintosh program development system.

The development of what was to become MPW started late in 1984, when Apple engineers designed a set of Macintosh programming tools for internal use. The name initially given to the package was the Macintosh Programming System, or MPS—initials which, coincidentally or otherwise, also stand for the last names of the three software engineers who developed it: Meyers, Parrish, and Smith!

▶ Launching MPW

The first version of MPW, Version 1.0, was released by APDA (the Apple Programmer's and Developer's Association) in September 1986. It was designed to work on any Macintosh with 1 MB of RAM and at least 1.6 MB of disk space. It had a shell that had been ported from the original Macintosh Development System and a C compiler that had been ported from the Lisa. But it also included a new 68000 assembler that had been developed from scratch. Other new features included the utilities Make and Print; the MacsBug debugger; and a pair of resource management tools called Rez and DeRez.

Version 2.0 of MPW, released in July 1987, included some new tools, an improved shell, an expanded MacsBug debugger, compilers that generated code for Motorola's new 68020 and 68030 chips, and new sets of interface and library files to support the Macintosh II. It was shipped on 800K floppy disks, and it required the use of the Mac Plus with 128K ROM and a hard-disk drive.

▶ MPW 3.0

The newest major revision of MPW, Version 3.0, was released in early 1989. Version 3.0 was faster and easier to use than was its predecessor, and it was the first MPW version to exploit the features of MultiFinder. It featured a new source-level debugger called the Symbolic Application Debugging Environment, or SADE; a rewritten version of MacsBug (6.0); a new C compiler; a new project management tool called Projector; some added tools; and some updated libraries and interfaces. Also, an Installer disk was included for installing MPW from a set of diskettes.

In MPW Version 3.1, a number of bugs were fixed and new capabilities for some tools were added. Version 3.1 also included a CPlus command for compiling programs written in C++ as well as new interface files for C++ programs.

▶ MPW 3.2

MPW 3.2, despite its unimpressive version number (they didn't call it MPW 4.0), is an ambitious revision of the Macintosh Programmer's Workshop. In fact, it is the first MPW revision since Version 3.0 that has included more than minor bug fixes.

The most visible new feature introduced in MPW 3.2 is a split-screen feature that can divide the MPW Editor window into as many as 20 scrollable panes. Black lines called split bars and slide boxes appear in the Editor window's vertical and horizontal scroll bars, as shown in Figure 1-1. By dragging these split bars and slide boxes, you can split the Editor screen into as many as 20 scrollable, sizeable panes. Since each pane has a pair of scroll bars, you can scroll each pane to display a separate portion of the document in the window.

Another new feature of MPW is a Browser window, shown in Figure 1-2. You can use the Browser window to change directories, inspect the contents of directories, and move to premarked sections of a document.

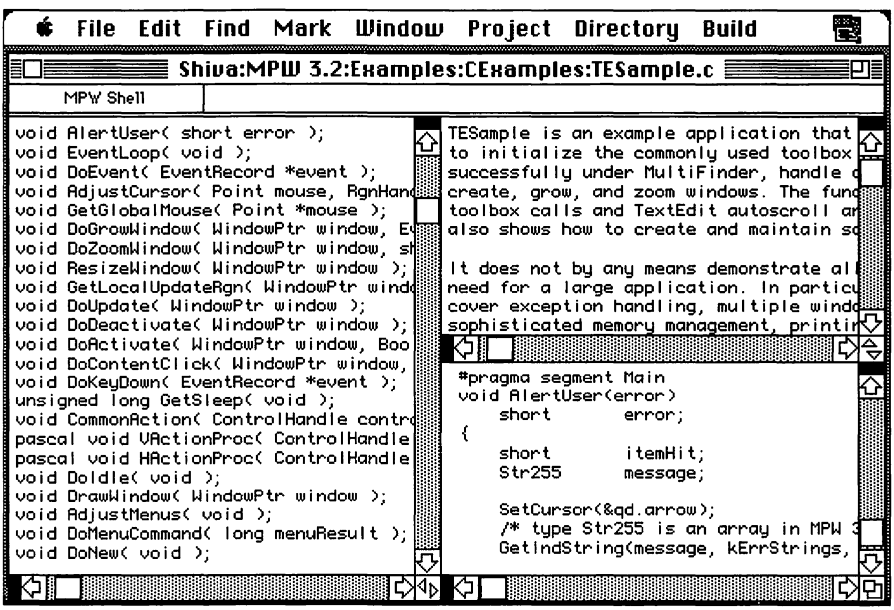


Figure 1-1. MPW 3.2 Editor

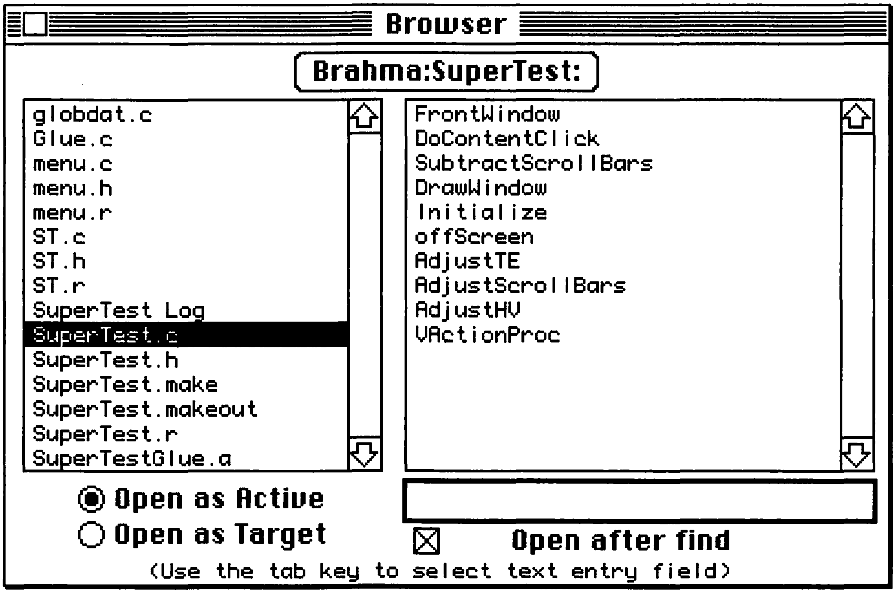


Figure 1-2. MPW 3.2 Browser

Both the split-window Editor and the MPW 3.2 Browser are described in more detail in Chapter 2.

Other new features added in MPW 3.2 include

- A new Object Pascal compiler and a new C compiler, which are available as separate products. The new C compiler supports the MacApp debugger and can produce "32-bit-clean" code to support the new, expanded memory capabilities of the Macintosh Memory Manager. The new Object Pascal compiler has more built-in support for MacApp and for external functions written in C.
- A non-interactive, script-driven text editor called StreamEdit. The StreamEdit tool, similar to the Sed tool used in UNIX, provides a method for editing and formatting documents automatically using stored scripts.
- Compatibility with the 32-bit addressing capability offered in System Software Version 7. With the release of Version 3.2, MPW is now 32-bit clean and can produce code that is 32-bit clean. (For more information about System 7's 32-bit addressing capabilities, see Chapter 7.)

- Updated versions of various libraries. The Runtime.o and CRuntime.o libraries have been merged into a single Runtime.o library, and the libraries have been resegmented to move more modules out of the "main" segment. The C libraries have been updated to conform to the current proposal for ANSI C, and the Pascal libaries have been enhanced to include standard C string functions that work on Pascal strings. (More information about libraries is presented in Chapter 8, "Building an Application.")
- New, speedier versions of the MPW tools Link and Lib, which link compiled programs and create object-code libraries. The Link and Lib tools are also covered in Chapter 8.
- Two new commands: ShowSelection, which scrolls a window to a selection and then finds and selects it; and SaveOnClose, which saves a window when the window is closed. (The syntax of these commands is in Appendix A.)
- Enhancements in several other tools and a number of bug fixes.

▶ What You Need to Run MPW 3.2

To run MPW 3.2, you must have at least a Macintosh Plus, a hard-disk drive, and 2 MB of RAM. In addition, you must be running System Version 6.0 or later, with either Finder Version 6.1 or later. If you want to use the SADE source-level debugger, you must use the System 7 Finder or MultiFinder, and you need at least 2.5 MB of RAM.

Those, of course, are the *minimum* requirements for running MPW, Apple recommends a system configuration of at least a Macintosh II equipped with 4 MB or more of memory and an 80 MB hard-disk drive. If your programming requirements are not too heavy, however, you can get by with a Macintosh SE, 4 MB of RAM, and a 20 MB hard-disk drive.

▶ Installing MPW 3.2

Installing MPW 3.2 is a snap; the MPW package now includes an installer disk, and the installation procedure is fully documented in Chapter 2 of the *MPW 3.0 Reference.* However, if you just cannot wait to get your MPW system up and running, you can follow these steps:

1. Make copies of all the master disks that came in your MPW packages, and put the original disks away for safekeeping. Use the copies that you have made for the following operations.

2. Insert your copy of the MPW Installation disk into your floppy-disk drive. Then, using the Finder or MultiFinder, drag the Installation Folder from your copy of the Installer disk on to the hard disk on which you want to install MPW.
3. Open the Installation Folder that is now on your hard disk, and double-click on the MPW Installer icon.
4. When the Installer program starts running, it prompts you to start inserting the copies of your MPW master disks into your floppy-disk drive. You can insert your MPW disks in any order, and you won't break anything if you insert a disk more than once.
5. Once installation is complete, you can throw away the Installation Folder (the one on your hard disk, not the one on your floppy), and you can then launch your newly installed shell.

Warning ▶

Use the MPW Installer. If you own an earlier version of MPW and want to update to MPW 3.2, be sure to use the MPW Installer script; don't try to install Version 3.2 by simply dragging the folders on the MPW master floppies onto your hard disk to replace your old ones. That is certain to cause you trouble because the contents of the folders on the MPW master disks changed with the release of MPW 3.0. Now, the files have been placed in folders that more closely reflect their final destination when they are moved to a hard disk.

In earlier versions of MPW, for example, the Pascal compiler was placed at the root level on the Pascal master disk. Now, Pascal is in a Tools folder on that disk. So, if you try to install MPW simply by dragging files and folders from a set of MPW master disks on to your hard disk, you'll wind up with conflicts between the old files and folders on your hard disk and the new files and folders that you drag over. And if your MPW system doesn't work right then, don't blame MPW.

▶ The Macintosh User Interface

From a user's point of view, some of the most important features that distinguish the Macintosh from less advanced computers are as follows.

- The mouse—The most important tool for manipulating the Macintosh cursor is the mouse—a pointing device which with a drag and a click can choose menu items; open, close, select, scroll, or resize windows; draw pictures and shapes; select locations where text will be typed or shapes will drawn; and cut, paste and copy text and graphics on the screen.

- Windows—All information displayed on the screen by a standard Macintosh application appears in windows. A window in which the user of an application can type text or draw shapes is called a document window. Windows can be equipped with various kinds of controls such as title bars, go-away boxes, zoom boxes, size boxes, and scroll bars. Windows can also contain buttons and icons, which a user can click on to perform various kinds of operations. The tasks that can be performed by clicking on buttons or icons are determined by the application that is running.

  More than one window can be displayed on the Macintosh screen, and windows can overlap each other. If the System 7 Finder or MultiFinder is running, windows from different applications can appear on the screen at the same time.

- Pull-down menus—When you run an application on the Macintosh, you do not have to make your way through various levels of menus to get from one part of the program to another. In a standard Macintosh program, the titles of all menus that you may want to access are displayed in a row at the top of the screen in a ribbon-shaped menu bar. To select an item from a menu, you simply press your mouse button in the title of the menu that you want. The title of the selected menu is then highlighted, and a column of menu items appears below it. You drag the mouse down to the menu item you want, and release the mouse button to select the chosen menu item.
- Dialogs—When an application needs more information from the user about a command, it can display a special kind of window called a dialog. Dialog windows, like document windows, can be equipped with various kinds of controls. When controls appear inside a dialog, they are known as dialog items.

  By clicking on button items that appear inside a dialog, or by typing text into a special kind of item called a TextEdit item, the user of an application can supply the application with whatever information it needs. In addition, dialogs can contain button items, icon items, and other kinds of items that can be defined by specific applications.

There are three kinds of Macintosh dialogs: modal dialogs, modeless dialogs, and alerts. Modal dialogs look just like windows, but contain controls; modeless dialogs have no title bar and are closed by clicking a button; and alert dialogs are modeless dialogs that display important messages.

- The Finder and MultiFinder—When you start up a Macintosh, the first screen you see is generated by a startup utility called the Finder. The Finder, contrary to what many people seem to believe, is not part of the Macintosh operating system; it is simply an application that is in the System file of a system disk and is launched when the system starts up. The Finder is responsible for presenting the unique desktop that you see when you start a Macintosh—a screenful of tiny icons representing disks, documents, file folders, and disk drives.

  In System Software Version 5.0, Apple introduced MultiFinder, an improved version of the Finder that allowed multiple applications to be opened simultaneously. With the introduction of System 7.0, the features of MultiFinder were integrated into the Macintosh operating system to provide what Apple calls "a cooperative multitasking environment."

  The System 7 Finder includes all of the features of MultiFinder, and several more. It supports color icons and miniature icons; has a stationery feature that lets the user create documents used as templates; and contains special folders for storing desk accessories and fonts, eliminating the need for the Font/DA Mover utility used in previous systems.

- Desk accessories—Desk accessories, or DAs, are mini-applications that can be started, used, and closed while larger applications are also running. If you have a desk accessory installed in your system, you can always select and run it, without leaving any other program that may be running. Menu items for all installed desk accessories always appear under the Apple menu on the Macintosh menu bar.

  With the introduction of System Software Version 7, the user has been given the option of treating any application as a desk accessory. You can now install a desk accessory simply by dragging its icon into the System Folder.

  System 7 also allows you to install fonts by dragging their icons into the System Folder. So the Font/DA Mover utility used in previous systems has become unnecessary.

Principles of Macintosh Programming

Since the Macintosh is a pretty unconventional computer, it should not be any surprise to learn that programming a Macintosh requires the use of some pretty unconventional programming techniques.

When you write a standard text-based program for a computer with a standard text-based operating system, you do not have to worry about such advanced user-interface features as mouse movements, windows, pull-down menus, or icons. When you write a program for a Macintosh, you do have to be concerned with handling all of these features—and more.

On the other hand, there are some ways in which writing a program for a Macintosh is actually easier than writing a program for a more conventional computer. When you develop an application for a non-Macintosh computer, for example, you usually have to have a fairly good understanding of the memory map of the computer you are working with; you have to decide exactly where in memory you are going to put your code, data, and screen graphics; and then you have to take all of the necessary steps to put each ingredient of your program in just the right memory location. Then, as your program grows, you have to reconfigure your computer's memory.

An Easier Way to Manage Memory

When you design an application for a Macintosh, you do not have to do any of that. In a Macintosh program, you will rarely, if ever, have to refer directly to the actual memory address of any block of code or data. That's because the Macintosh has a built-in Memory Manager, which, as its name implies, performs memory management functions. The Macintosh also has a number of other managers that are designed to handle other kinds of important procedures and operations.

Some of these managers—the Memory Manager among them—are built into the Macintosh operating system. Other managers are provided in the User Interface Toolbox, a collection of hundreds of useful routines that are provided with every Macintosh and can be used in any Macintosh program. Some portions of the Toolbox are built into ROM, and others are stored on the Macintosh system disk.

When a user loads a program into a Macintosh, the Memory Manager first decides exactly where each part of the program should be stored in memory, and then it places every piece of code and data in the program in its proper memory location. Then, as the program runs, the Memory Manager automatically shifts blocks of memory around to make room for new blocks as memory requirements change.

The Memory Manager takes care of all of this memory manipulation by using not only pointers, but also pointers *to* pointers, which are called handles. By using handles in your Macintosh programs, you can let the Memory Manager worry about the physical memory locations of all the data that you refer to in your code, and you will never again have to refer to any block of code or data by its actual memory address. Much more information about the Memory Manager appears in Chapter 7.

▶ Macintosh I/O

File management is another programming headache that you need not worry about when you're writing a Macintosh program. That's because the Macintosh is equipped with a Standard File Package, which takes care of such jobs as finding directories and opening, closing, and saving files.

When you write a Macintosh program that gives the user the option of loading or saving a file, all you have to do is call the Standard File Package. The Standard File Package then displays a dialog—or a series of dialogs—that allow the user to locate any desired directory on any disk and then to load or save the selected file. Therefore, you can avoid a lot of I/O hassles by using the Standard File Package.

▶ Managing Resources

Another important manager in the Toolbox is the Resource Manager —which, as its name implies, handles the resources that a Macintosh program uses. Resources are blocks of static data such as menus, dialogs, window templates, and cursors. They are created, stored, and manipulated separately from a program's code for flexibility and ease of maintenance. The Resource Manager is covered in much more depth in Chapter 6.

▶ QuickDraw and Macintosh Graphics

When you type text or draw graphics on the Macintosh screen, all drawing operations are handled by a very important part of the Toolbox called QuickDraw. QuickDraw is the heart of the Macintosh graphics system. Whether you want to draw into a window or just set up a simple shape such as a rectangle to be called by other managers in the Toolbox, your applications will usually make calls to QuickDraw.

Although not every version of the Macintosh has been equipped to handle color, every version of QuickDraw has supported both black-and-white graphics and a limited capability of producing images in up to 16 colors. Beginning with the Macintosh II, an enhanced version of QuickDraw, supporting up to 2^{48} colors, has been available. This newer version of QuickDraw is called, logically enough, Color QuickDraw. In this book, QuickDraw is mentioned only as it relates to MPW programming. More comprehensive information about QuickDraw and Color QuickDraw can be found in *Inside Macintosh*, Volumes I and IV.

▶ Important Events

Before we end this summary of Macintosh features and open up the User Interface Toolbox, it is important to mention a programming technique called event-driven programming. Every program written in accordance with the Macintosh User Interface Guidelines contains a main event loop, a loop that constantly monitors such user actions as mouse clicks and the use of keys on the Macintosh keyboard. When an application user clicks the mouse or presses a key, that action is known as an event, and it is up to the application to detect the event and respond to it appropriately.

To help programs manage events, the Macintosh Toolbox has been supplied with a manager called the Event Manager, and the Macintosh operating system contains a set of calls referred to as the Operating System Event Manager. Both the Event Manager and the OS Event Manager are covered in greater detail in Chapter 5.

▶ Other Features

The Macintosh also has many built-in features that can help you perform such tasks as tracking mouse movements and mouse clicks and can assist you in such jobs as drawing and manipulating windows and dialogs, and building and manipulating pull-down menus. Some of these tools are built into the Macintosh Toolbox, and others are built into the Macintosh operating system.

▶ The Macintosh Toolbox and Operating System

In the preceding sections, we have mentioned three features that distinguish the Macintosh from more conventional computers: the Macintosh User Interface, the User Interface Toolbox, and the Macintosh operating system. Now let's put these three features together and see how they work together. We will start at the lowest level of processing: the operating system level.

When an application is running on a Macintosh, the portion of code that communicates most directly with the central processor is the operating system; its job is to perform basic operating tasks such as input and output, memory management, and interrupt handling.

One level above the operating system lies the User Interface Toolbox, which was designed to help programmers implement the standard Macintosh User Interface in their applications easily and efficiently. When you call a Toolbox routine in an application, the Toolbox often calls an operating system routine when it wants to perform a low-level operation. When you write programs for the Macintosh, you will often bypass the Toolbox and call the operating system directly in your applications.

Applications, as well as other kinds of programs written for the Macintosh, lie one level above the User Interface Toolbox. Well-behaved programs—a term that Apple often uses to describe programs written in accordance with its User Interface Guidelines—perform most of their essential tasks by making calls to the Toolbox and the operating system.

At the very top of the processing hierarchy is the User Interface, which, as its name indicates, is the interface between the Macintosh and the user of a program. Windows, menus, dialogs, and controls—and such specialized applications as the System 7 Finder and its predecessor, MultiFinder—are all parts of the User Interface, as you have seen in earlier sections of this chapter.

The four levels of Macintosh processing—the operating system, the Toolbox, applications, and the User Interface—are illustrated in Figure 1-3.

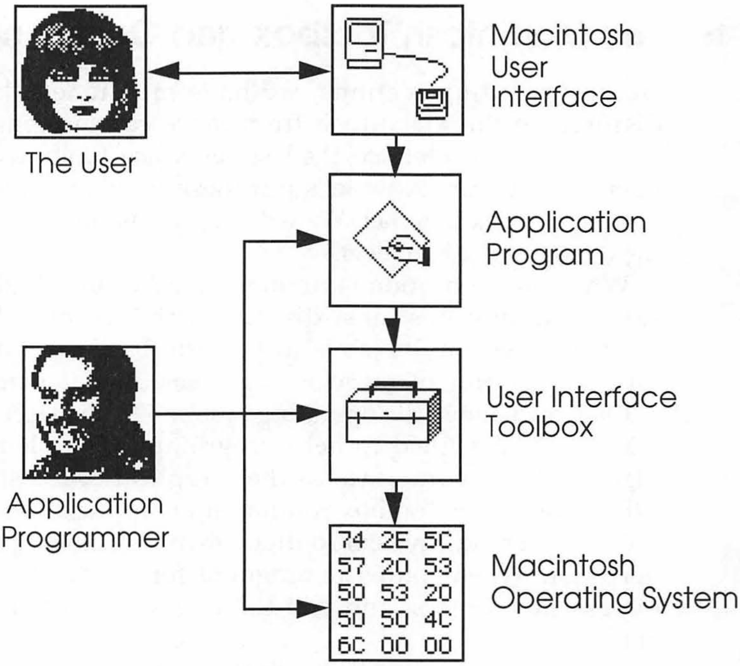


Figure 1-3. The four levels of program processing

▶ The User Interface Toolbox

The User Interface Toolbox is a collection of hundreds of routines and functions that you can use in your programs without having to write all of the code that they contain from scratch. The routines in the Toolbox —like the libraries of C and Pascal functions that programmers of conventional computers often purchase and use—are prewritten, pretested procedures and functions that can be incorporated into programs to perform specific tasks. But, unlike the "cookbooks" of routines that programmers of less advanced computers so often use, the procedures and functions in the Macintosh Toolbox are always available, free of charge, and are specifically designed to work correctly with the Macintosh User Interface, the Macintosh architecture, and the Macintosh operating system. Furthermore, since they are written and maintained by the people who designed your Macintosh, they are guaranteed to work properly not only with the computer you are currently using, but also with future models.

Most of the functions and procedures in the Toolbox are designed to help you implement the Macintosh User Interface—the windows, pull-down menus, dialogs, and standard control mechanisms mentioned earlier in this chapter.

▶ The System 7 Toolbox

Until System 7 was unveiled, the number of managers in the Macintosh Toolbox had grown steadily but slowly. With the introduction of System 7, Apple pulled out all the stops and added eight new managers. These new managers are illustrated in Figure 1-4 and described in this section.

▶ The Process Manager

The Process Manager manages the scheduling of processes that affect open applications and desk accessories. Under System Software Version 7, any application can be placed under the Apple menu and used as a desk accessory, and the number of processes is limited only by available memory.

With the help of the Process Manager, multiple applications running under System 7 can share the 680X0 microprocessor and other resources. The Process Manager provides applications with a means of sharing the amount of memory available, and also sharing access to the CPU.

In addition to managing the scheduling of applications, the Process Manager manages access to shared resources and loads applications into memory. By querying the Process Manager, an application can get

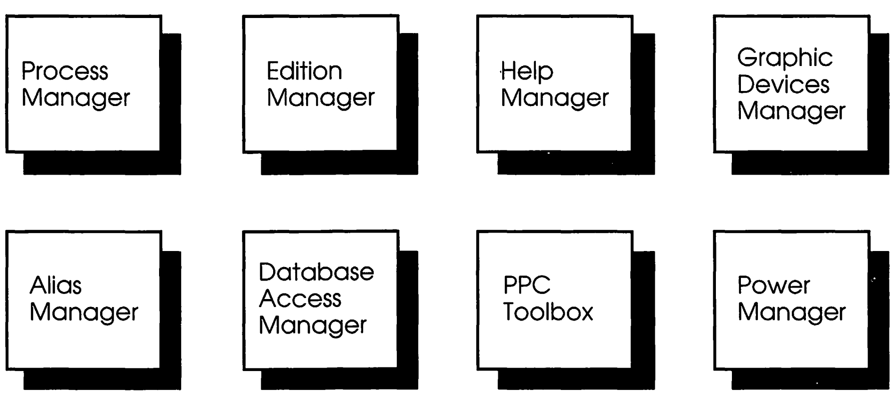


Figure 1-4. System 7 Toolbox managers

information about itself or any other open application, such as the number of free bytes in the application's heap.

The System 7 Finder, which carries out actions directed by the Process Manager, is shown in Figure 1-5.

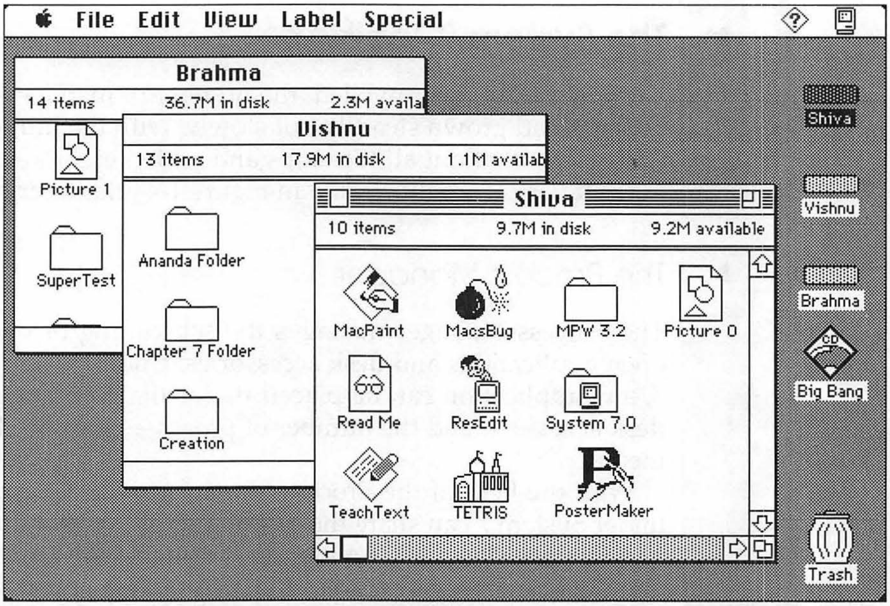


Figure 1-5. System 7 Finder

▶ The Edition Manager

The Edition Manager allows applications and documents to share data dynamically and also allows users to share data dynamically across a network.

With the Edition Manager, the Macintosh user can

- capture data from a document and integrate it into another document
- modify information in a document, simultaneously updating any document that shares its data
- share information between applications on the same disk or across a network of Macintosh computers

The Edition Manager's functions are similar to the standard cut, copy, and paste features offered since the advent of the first Macintosh. With the help of the Edition Manager, however, text and graphics that are edited in one application can also change in any associated applications that may be running—either on the same computer or on a network. Text, graphics, spreadsheet cells, database records—any data that can be selected within an application—is accessible to other applications supporting the Edition Manager.

▶ The Help Manager

The Help Manager can display cartoon-like help balloons when the user of an application moves the mouse into a user interface element such as a menu, a window, an icon, or a control.

In an application that makes use of the Help Manager, the user can enable help balloons by choosing "Show Balloons" from the Help menu. The contents of the help balloons are provided by the application. The user can turn off the help function by selecting "Hide Balloons" from the Help menu.

Figure 1-6 shows a help balloon created by the Help Manager.

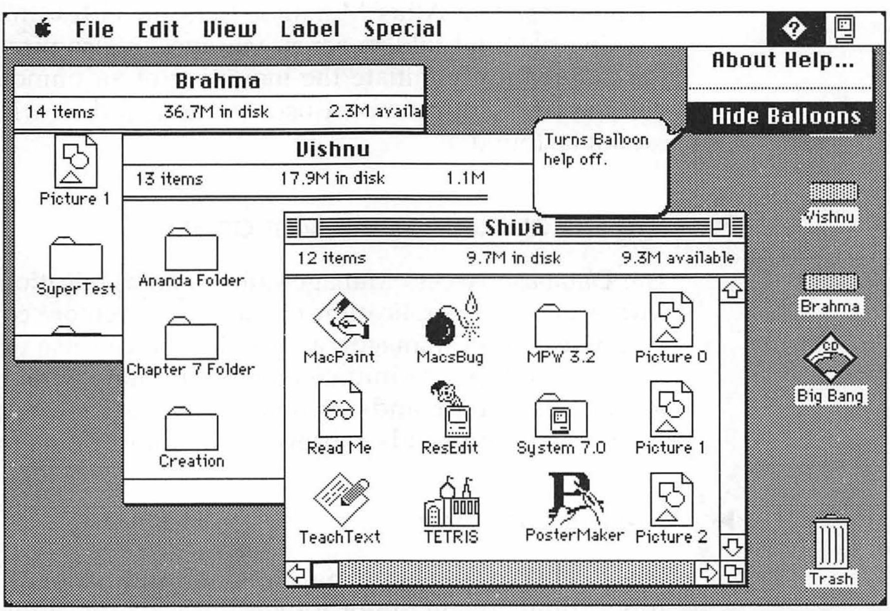


Figure 1-6. Help balloon

The Graphics Devices Manager

The Graphics Devices Manager manages offscreen graphics. With the Graphics Devices Manager, you can create images offscreen and then move them quickly into view with a single routine. This technique prevents the jumpiness that you sometimes see when you draw object-oriented graphics directly on the screen. Also, by drawing a picture in an environment that you create and control, you can be sure that no other application or desk accessory changes its characteristics.

The Graphics Devices Manager also contains routines and data structures used by QuickDraw and the Palette and Color managers to communicate with the graphics devices attached to a particular system. Such devices may include printers as well as video screens. Most of these routines are used only by the operating system; some may be used by graphics-intensive applications.

The Alias Manager

The Alias Manager stores file and directory information in specially designed records called *alias records*. Files and folders with alias records can be referred to later by their aliases, rather than by their full pathnames. The Alias Manager thus provides an easy method for tracking files and folders across volumes. It also provides routines that can automatically initiate the mounting of an unmounted AppleShare volume, and can prompt a user to insert a disk when a needed disk cannot be found.

The Database Access Manager

The Database Access Manager allows an application to communicate with a database application running on a remote computer. With the Database Access Manager, an application can use either high-level or low-level routines to initiate communications with a remote database server; send commands or data to the server, and, after the server executes the commands, retrieve any requested data from the server.

The PPC Toolbox

The PPC (Program-to-Program Communications) Toolbox enables applications to communicate with other applications. This low-level manager is most suitable for code modules (or desk accessories or applications) that are not event-driven.

With the PPC Toolbox, an application can

- verify the identities of remote users of the PPC Toolbox
- share information among other applications running on the same computer or on a computer network

▶ The Power Manager

The Power Manager, used only by the Macintosh Portable, is built into the computer's firmware. The Power Manager can put the Macintosh Portable into two low-power-consumption states: the idle state and the sleep state.

The Macintosh Portable goes into its idle state when the system has been inactive for 15 seconds. When the computer is in the idle state, its normal 16MHz clock speed is slowed down to 1MHz.

When the portable has been inactive for an additional period of time—the duration is set by the user—the computer's power is shut off, but no data is lost from RAM. When the user activates the computer, by clicking the mouse button or pressing a key, the portable "wakes up" and is ready for action.

▶ Other Toolbox Managers

Figure 1-7 shows the managers that made up the Toolbox prior to the introduction of Software System Version 7. Their dependencies on each other are illustrated roughly by their position on the chart; managers that are lower on the chart often call the upper ones. However, their precise dependencies are too complex to be illustrated in a simple diagram.

Managers included in the pre-System 7 Toolbox are listed in Table 1-2. Three managers—the Toolbox Event Manager and the Resource Manager from the Toolbox and the Memory Manager from the operating system—are so important that they have their own chapters in this book: Chapters 5, 6, and 7.

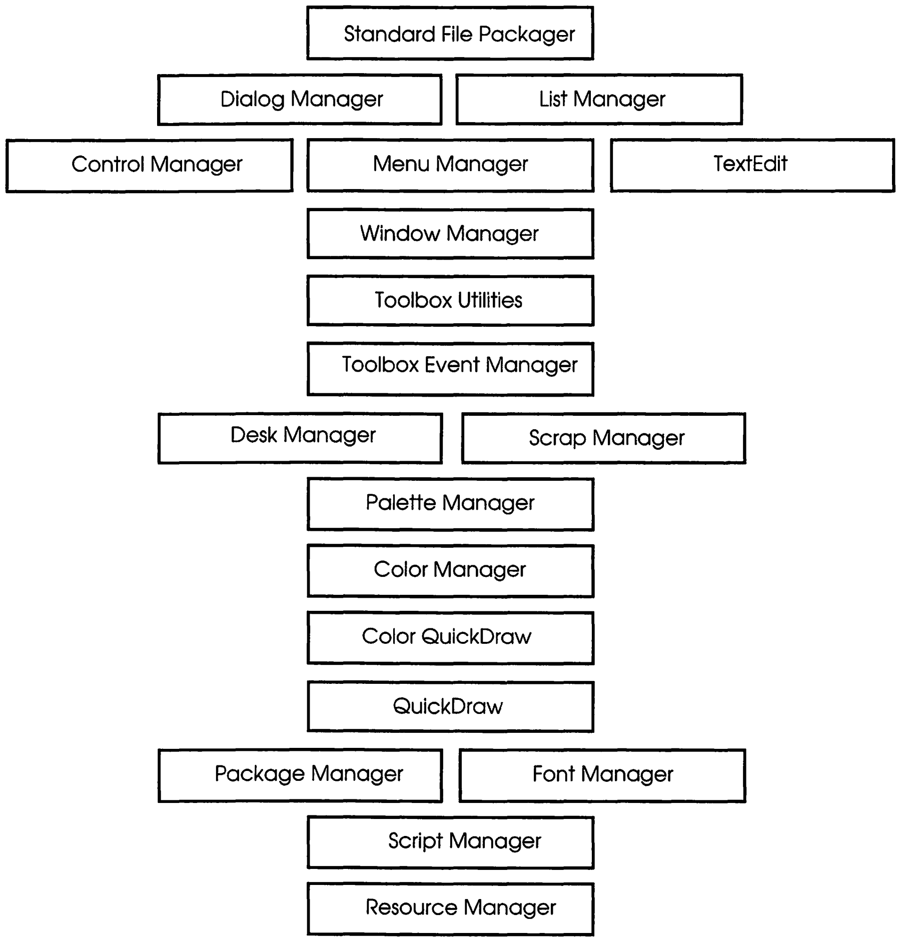


Figure 1-7. The System 6 Toolbox

Table 1-2. The System 6 Toolbox

Manager	*Description*
Toolbox Event Manager	Often referred to simply as the Event Manager, the Toolbox Event Manager reports events, such as mouse clicks and key presses, to an application. The application, in a main event loop, determines what to do about each event reported. In System 7, some new kinds of events are recognized. For more details, see Chapter 5.
Window Manager	Takes care of all document windows displayed on the Macintosh screen. Provides routines that create, open, close, resize, and move windows around on the screen.
Menu Manager	Sets up and manages the menus and menu items on the Macintosh menu bar.
Control Manager	Creates and manages controls, such as buttons, check boxes and scroll bars, inside windows and dialogs.
Dialog Manager	Creates and displays modal and modeless dialog and alert windows, and monitors the user's responses to dialog items.
Resource Manager	Manages and keeps track of the resources used by a program. Resources are blocks of static data, such as menus, dialogs, window templates, and cursors, which are created, stored and manipulated separately from a program's code for flexibility and ease of maintenance. In System 7, some new kinds of resources have been added. The Resource Manager is the topic of Chapter 6.
QuickDraw	The heart of the Macintosh graphics system. QuickDraw performs all drawing operations on the screen, including both graphics and text. QuickDraw can handle both black-and-white images and images with up to 16 colors.
Color QuickDraw	A greatly enhanced version of QuickDraw, Color QuickDraw, was introduced with the Macintosh II. Color QuickDraw is capable of displaying up to 2^{48} colors on a screen.

Table 1-2. The System 6 Toolbox (continued)

Manager	*Description*
Color Manager	Provides color-selection support for Color QuickDraw by giving applications a consistent method for producing color displays on the Macintosh II and other models of the Macintosh that offer advanced color capabilities.
Palette Manager	Responsible for monitoring and establishing the color environment of the Macintosh II and other models of the Macintosh with advanced color capabilities. Includes procedures and functions to manage shared resources, as well as providing an enormous selection of colors for programs that demand more colors than Color QuickDraw's default selections can provide.
Font Manager	Supports the drawing of text by QuickDraw. Before QuickDraw draws text, it calls the Font Manager, which does the background work necessary to make a variety of character fonts available in various sizes and styles. In System 7, the Font Manager supports outline fonts, which eliminate jagged edges from displayed and printed characters, regardless of their size.
TextEdit	Provides applications with a means of accessing user input via the keyboard. TextEdit displays text typed by the user, and automatically provides applications with cutting, pasting, and copying capabilities via a standard Macintosh utility called the Clipboard. Since the introduction of Software System Version 6.0, TextEdit has also been capable of handling text styling.
Scrap Manager	Supports the use of the Clipboard, a built-in utility for cutting, copying, and pasting text or graphics within a single program or between programs.
Script Manager	Enables applications to function correctly with non-Roman writing systems, or scripts, such as Japanese, Chinese, or Arabic, as well as with roman-based writing systems such as English.

Table 1-2. The System 6 Toolbox (continued)

Manager	*Description*
Standard File Package	Displays a standard User Interface dialog for locating and specifying a document file and handles file I/O by calling a lower-level operating system package, the OS File Manager.
Package Manager	Supports the use of several special pieces of system software called packages. The List Manager is one manager that is stored as a package. Two packages are extensions to the Toolbox Utilities manager. They are the Binary-Decimal Conversion Package, which converts integers into decimal strings and vice versa, and the International Utilities Package, which can be used to make applications independent of country-specific information by providing such details as the formats for numbers, currency, dates, and times.
List Manager	Supports the use of lists by applications. Lists handled by the List manager can be stored as one- or two-dimensional arrays, and can be sorted, displayed, and scrolled.
Desk Manager	Supports desk accessories, small programs that can be run from within an application. The user opens desk accessories by choosing an item from the Apple menu. With the introduction of System Software Version 7, it has become possible to use any application as a desk accessory.
Toolbox Utilities	A collection of miscellaneous utilities, including managers that handle fixed-point arithmetic, string manipulations, and logical operations on bits.

▶ The Macintosh Operating System

The operating system, as mentioned previously, is at the lowest level of the Macintosh user interface hierarchy; it performs basic tasks such as input and output, memory management, and interrupt handling.

The User Interface Toolbox is a level above the operating system; it was designed to help programmers implement the standard Macintosh user interface in their applications. The Toolbox calls the operating system to do low-level operations, and you can also call the operating system directly.

The most important operating system managers are shown in Figure 1-8 and listed in Table 1-3.

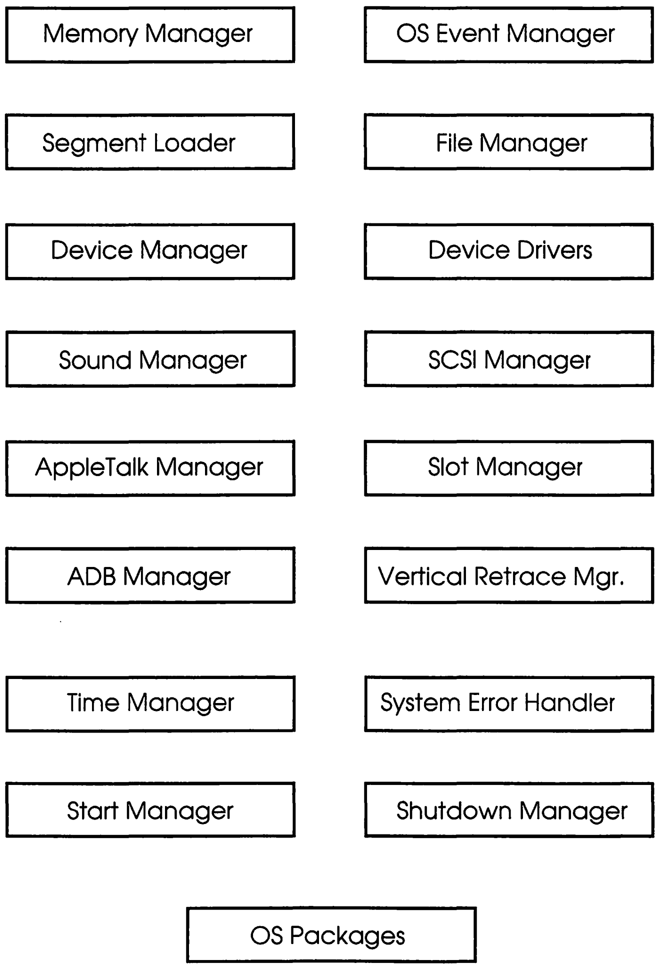


Figure 1-8. Operating system managers

Table 1-3. Operating system managers

Manager	*Description*
Memory Manager	Dynamically allocates and releases memory used by applications and other parts of the operating system. In Macintosh programs, memory space is obtained by calls to the Memory Manager. In System 7, new capabilities have been added to the Memory Manager. They include virtual memory, temporary memory, and 32-bit addressing. More information about the Memory Manager is presented in Chapter 7.
OS Event Manager	The Operating System Event Manager, or OS Event Manager, reports low-level, hardware-related events, such as mouse clicks and keystrokes. The OS Event Manager is normally called by the Toolbox Event Manager. Both Event Managers are covered in detail in Chapter 5.
Segment Loader	Loads pieces of an application's code into memory to be executed. The Segment Loader also serves as a bridge between the Finder and an application, letting the application know whether it has to open or print a document when it starts up. For more information on the Segment Loader, see Chapter 7.
File Manager	Contains low-level I/O routines that handle communications between an application and files on block devices such as disk drives. The Standard File Package calls the File Manager when it needs to perform such tasks as locating, loading, and saving files. Applications can also call the File Manager.
Device Manager	Handles communication between applications and devices. A device is a piece of external equipment, or part of the Macintosh itself, that can transfer information into or out of the computer. Devices include disk drives, serial communications ports, sound and music generators (on the Macintosh Plus), video drivers (on the Macintosh II and later models), and printers.

Table 1-3. Operating system managers (continued)

Manager	*Description*
Device Drivers	Handle the task of making various kinds of devices present the same kind of interface to an application. Three drivers are built into ROM: the Disk Driver, the Sound Driver, and the Serial Driver. Several other drivers, including the Printer Driver and the Video Driver, are in RAM. Device drivers also handle the operations of desk accessories.
Sound Manager	Supports sound and music on the Macintosh II and later Macintosh models. The Sound Manager has been greatly improved in System 7.
SCSI Manager	Supports the Small Computer System Interface (SCSI), an interface for hard-disk drives and other high-speed peripheral devices.
AppleTalk Manager	Provides an interface to a set of AppleTalk drivers that enable programs to send and receive information over an AppleTalk network.
Slot Manager	On the Macintosh II and later models, the Slot Manager enables programs to communicate with expansion cards in NuBus slots.
ADB Manager	Supports the Apple Desktop Bus, a hardware device used for connecting low-speed input devices, including the mouse and keyboard, to the Macintosh. The ADB Manager was not a part of the operating system until the introduction of the Macintosh II and the Macintosh SE.
Vertical Retrace Manager	Handles the scheduling and execution of tasks during the vertical retrace interval, the period of time during which the Macintosh hardware generates a vertical retrace interrupt. A vertical retrace interrupt, sometimes referred to as the system "heartbeat," takes place 60 times every second. For compatibility purposes, the heartbeat rate of a small-screen Macintosh is emulated by the Macintosh II and other large-screen models.

Table 1-3. Operating system managers (continued)

Manager	*Description*
Time Manager	Provides a hardware-independent means of timing program operations. Greatly enhanced in System 7.
System Error Handler	Assumes control if a system error occurs. If that happens, the dreaded "bomb" dialog containing an error message is displayed, and the System Error Handler provides a mechanism for the user either to restart the system or attempt to resume execution of the application.
Start Manager	Orchestrates all activities related to system testing and startup.
Shutdown Manager	Provides the user with a mechanism for restarting the Macintosh or shutting it off.
OS Packages	Three OS Packages perform low-level operations: the Disk Initialization Package, which the Standard File Package calls to initialize and name disks; the Floating-Point Arithmetic Package, which contains a random-number generator and also supports extended-precision arithmetic according to Standard 754 of the Institute of Electrical and Electronics Engineers (IEEE); and the Transcendental Functions Package, which contains trigonometric, logarithmic, exponential, and financial functions. The Floating-Point Arithmetic Package and the Transcendental Functions Package support the Standard Apple Numerics Environment (SANE).

▶ Unlocking the Toolbox

As wonderful as they are, the hundreds of routines in the Toolbox and the operating system would not do anybody much good if there weren't a quick and easy way to get to them from a program. Fortunately, Apple has made it just about as easy to call a Toolbox or operating system procedure as it is to call any other procedure in a program.

Toolbox and operating system calls are often lumped together and referred to as trap calls, or simply traps. Their name stems from the fact that the central processor in Macintosh intercepts calls Toolbox and operating system procedures using a feature of the 680X0 processor known as the 1010 emulator trap. The calls are then made by a set of procedures known collectively as the trap dispatch system.

The trap dispatch system was devised so that programs written for the Macintosh could make Toolbox and operating system calls without having to jump to the routines' physical memory locations. By eliminating the need to access Toolbox calls by their actual memory addresses, Apple gave its engineers a way to change the physical locations of Toolbox calls in new versions of the Macintosh ROM without making old applications obsolete or affecting the way in which new applications would have to be written.

To accomplish this goal, the designers of the Macintosh created a trap dispatch table that contained the addresses of all Toolbox and operating system routines. This table was stored in low memory. Then a system was devised to use the encoded addresses in the trap dispatch table to make Toolbox and operating system calls. That way, an application could make a Toolbox call by using the trap dispatcher rather than by jumping to the routine's actual address. That meant that the addresses of Toolbox calls could be moved around in memory by Apple's development engineers, as long as the information in the trap dispatch table was kept up to date.

As the Macintosh has evolved and has become more and more sophisticated, the wisdom of having taken this approach has been proven over and over again. Since the unveiling of the original Macintosh, addressable RAM has grown from 128K into the one-gigabyte range. The sizes of both the Toolbox and the operating system have increased by leaps and bounds, with new calls—and even whole new managers—being steadily added in ever-growing numbers. To hold the addresses of all these new calls, the trap dispatch table has been expanded, and memory addresses of many new traps have been added.

Furthermore, although most Toolbox routines are in ROM, some are in RAM. Still others have been "patched," that is, altered to eliminate bugs or to be compatible with new models, and many of these patched calls reside partly in ROM and partly in RAM! Today, as new Toolbox and operating system calls are introduced, they usually make their first appearance on a system disk and are not moved into ROM until Apple is certain that they are bug-free and are coded as compactly and as efficiently as possible.

▶ How the Trap Dispatcher Works

On the Macintosh, all ROM calls are written as single 680X0 instructions. Because of the way the 680X0 processor is designed, no valid instructions begin with the hexadecimal digit A. Therefore the designers of the Macintosh decided to use the instructions $A000 through $AFFF to emulate actual 680X0 instructions: that is, to use them to provide access to Toolbox and operating system routines.

When the microprocessor sees an instruction that begins with the hexadecimal digit A, it immediately recognizes the instruction as invalid, or as an unimplemented instruction. So it creates a 68000 exception and jumps to a routine whose memory address is stored at a certain location—specifically, address $28. This address, called an exception vector, turns control over to the trap dispatcher.

The trap dispatcher, by looking at the portion of the word that follows the hex number A, determines the address of the routine to be called by getting it from the trap dispatch table. Once it has looked up the address, it uses the machine-language instruction JSR (jump to subroutine) to jump to the appropriate Toolbox call.

▶ Calling the Toolbox from MPW

To make a Toolbox or operating system call from a program written using MPW, you do not really have to be concerned about how the trap dispatch system works. That's because the MPW C compiler, the MPW Pascal compiler, and the assembler all come with sets of interface files that can be accessed from programs to make Toolbox and operating system calls. The interface files for the C compiler are in a folder called CIncludes. The interface files for the MPW assembler are in a file called AIncludes. And those for the Pascal compiler are in a folder called PInterfaces.

The source-code fragments in the following examples were put together to give you a general idea how trap calls are handled in MPW C, MPW Pascal, and MPW assembly language. In later chapters, we'll use similar procedures to write, compile, and link complete programs.

By the Way ▶

Routines, Procedures, and Functions. In Pascal, there is a sharp distinction between a function and a procedure. If a routine returns a value, it's a function; if it doesn't, it's a procedure.

In C, no distinction is made between a function and a procedure. Whether a routine returns a value or not, it's still a function.

Since this book makes references to both Pascal and C, the terms routine, procedure, and function are used somewhat interchangeably. But I have tried to make sure that the differences in their meanings are made clear from their context.

▶ Calling Traps in C

Inside the C compiler's CIncludes folder, there is a large set of header files, one for each manager in the Toolbox and the operating system. As you would expect, each header file ends with C's standard ".h" extension. It's easy to figure out which header file covers which manager because the name of each file corresponds (though not always exactly) to the name of the manager that it handles. The header files in the CIncludes folder are listed in Table 1-4.

Table 1-4. MPW C header files

ADSP.h	DDEV.h
Aliases.h	Desk.h
AppleEvents.h	DeskBus.h
AppleTalk.h	Devices.h
Assert.h	Dialogs.h
Balloons.h	DisAsmLookup.h
CommResources.h	DiskInit.h
complex.h	Disks.h
Connections.h	Editions.h
ConnectionTools.h	EPPC.h
Controls.h	ErrMgr.h
CRMSerialDevices.h	errno.h
CTBUtilities.h	Errors.h
CType.h	Events.h
CursorCtl.h	FCntl.h
DatabaseAccess.h	Files.h

Table 1-4. MPW C header files (continued)

FileTransfers.h	QDOffscreen.h
FileTransferTools.h	Quickdraw.h
FixMath.h	Resources.h
Float.h	Retrace.h
Folders.h	ROMDefs.h
Fonts.h	SANE.h
fstream.h	Scrap.h
generic.h	Script.h
GestaltEqu.h	SCSI.h
Graf3D.h	SegLoad.h
HyperXCmd.h	Serial.h
IOCtl.h	SetJmp.h
iomanip.h	ShutDown.h
iostream.h	Signal.h
Limits.h	Slots.h
Lists.h	Sound.h
Locale.h	StandardFile.h
Math.h	Start.h
Memory.h	StdArg.h
Menus.h	StdDef.h
MIDI.h	StdIO.h
new.h	stdiostream.h
Notification.h	StdLib.h
OldStream.h	stream.h
OSEvents.h	String.h
ostream.h	Strings.h
OSUtils.h	strstream.h
Packages.h	SysEqu.h
Palette.h	Terminals.h
Palettes.h	TerminalTools.h
Perf.h	TextEdit.h
Picker.h	Time.h
pipestream.h	Timer.h
PLStringFuncs.h	ToolUtils.h
Power.h	Traps.h
PPCToolbox.h	Types.h
Printing.h	Values.h
PrintTraps.h	Video.h
Processes.h	Windows.h

One of the header files in the CIncludes folder is called Windows.h. As its name implies, the Windows.h file contains header definitions that are used to make calls to the Window Manager.

To use the Windows.h file in a C program, you must include the name of the file at the beginning of the program with a line like this:

```
#include <Windows.h>
```

Then you must follow this calling convention:

```
pascal void CloseWindow(WindowPtr theWindow)
```

In this example, the WindowPtr argument that is passed to the CloseWindow function is a pointer to a data structure that is declared in the Windows.h header file as WindowRecord. In the Windows.h header file, a WindowRecord structure is declared in this way:

```
struct WindowRecord {
    GrafPort port;
    short windowKind;
    Boolean visible;
    Boolean hilited;
    Boolean goAwayFlag;
    Boolean spareFlag;
    RgnHandle strucRgn;
    RgnHandle contRgn;
    RgnHandle updateRgn;
    Handle windowDefProc;
    Handle dataHandle;
    StringHandle titleHandle;
    short titleWidth;
    ControlHandle controlList;
    struct WindowRecord *nextWindow;
    PicHandle windowPic;
    long refCon;
};

typedef struct WindowRecord WindowRecord;
typedef WindowRecord *WindowPeek;
```

The first field in a WindowRecord is a GrafPort, a data structure used by QuickDraw to draw on the screen. The other fields in the WindowRecord control various characteristics of windows.

In the Windows.h header file, the CloseWindow function itself is defined as being of type pascal because all calls in the Macintosh Toolbox and operating system use what are known as Pascal-compatible calling conventions; that is, they pass their parameters to the Toolbox and the operating system as if they were written in Pascal rather than in C. Specifically, the function-calling conventions that MPW C and Pascal use differ in the order of parameters on the stack, the type of coercions that are applied to the parameters, the method of storing the returned result, and the number of microprocessor scratch registers used. Further information about how Pascal and C calling conventions differ can be found in the *Macintosh Programmer's Workshop C 3.0 Reference* and in the *Macintosh Programmer's Workshop Pascal 3.0 Reference.*

Calls to the MPW Toolbox follow Pascal calling conventions because the Toolbox was originally designed to work with Pascal compilers. But that does not mean that you must use Pascal-style calling conventions for functions that you write in MPW C; in functions that you write for your own programs, you can use the calling conventions of standard C.

Furthermore, you'll probably never even notice that Toolbox and operating system calls use Pascal-style calling conventions. The calls are all defined in the MPW C compiler's header files, so you won't have to worry about how they are defined when you write C programs. All you have to do is call any function you need, in exactly the same way you would call any other function. To make a CloseWindow call in a C program, for example, all you have to do is type

```
CloseWindow(window);
```

and the MPW C compiler ensures that the call is passed to the Toolbox correctly.

Starting Up Tools in C

When you have included all of the header files you need in an MPW C program, you must make sure that the Toolbox and operating system managers that make the calls are initialized. Some managers, such as the Memory Manager and the Resource Manager, are initialized automatically at boot time and do not have to be specifically initialized in application programs. However, other managers *do* have to be initialized.

Since some Toolbox and operating system managers call other managers to perform certain operations, the order in which you initialize the various managers is significant. For example, before you can use the ToolBox Event Manager, you must initialize the Window Manager if you use window operations in your program. Before you initialize the the Window Manager, you must initialize both QuickDraw and the Font Manager. You must also initialize QuickDraw before you can initialize many other parts of the Toolbox.

You could sit down and work out a dependency list that could tell you at a glance the exact order in which all Macintosh managers must be initialized. But that is not really necessary. Since some managers are initialized automatically, and since most Macintosh programs call most of the standard managers—QuickDraw, the Window Manager, the Control Manager, the Dialog Manager, and so on—the easiest way to initialize the managers you are most likely to need is to find a program that contains a well-behaved initialization segment—and copy it!

There's nothing wrong with copying; in fact, it's encouraged. Inside your MPW folder, there's a folder called Examples, and in that folder there are sample programs written in C, Pascal, and assembly language, as well as HyperCard externals and sample code to help you use Projector. All of these examples were included in the MPW package for you to use—by studying them or by copying parts of them into your own programs. For example, the following piece of code is from a program called TESample.c that is in the MPW Examples folder:

```
InitGraf((Ptr) &qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
```

In this code fragment are startup calls for QuickDraw, the Font Manager, the Window Manager, the Menu Manager, TextEdit, and the Dialog Manager—and they are all started up in the right order.

Compiling and Linking a C Program

When you have written a program in MPW C, you must compile it using the MPW C compiler, and then link it using a tool called the MPW Linker.

The Linker is usually invoked with a special kind of MPW script called a makefile. A makefile is an MPW text file, or script, that contains instructions for building, or converting, a source-code program into an executable object-code program. A makefile describes the dependencies between the components of the program, along with the shell commands needed to build each component. By executing the commands created by the makefile, you can build the program.

You can create a makefile by pulling down and selecting the Create Build Commands item under the Build menu on the MPW menu bar, or you can write your own makefile. Rules for writing makefiles are explained in Chapter 8. Once you have created a makefile, you can run it by executing the MPW command Make, which is also described in Chapter 8.

One of the functions of a makefile is to link the object code generated by an MPW compiler (or the MPW assembler) to a set of object-code libraries that are needed to make Toolbox and operating system calls. These libraries reside in an MPW folder called Libraries.

To link the object code of a compiled C program to the libraries that are needed to make Toolbox and operating system calls, you must include the appropriate linking commands in your makefile. For example, the following block of code includes links to the C libraries Runtime, StdLib, and CInterface, as well as to the MPW libraries Interface.o and ToolLibs.o.

```
Link {SymOptions} -w -c 'MPS ' -t MPST MyProg.c.o
FStubs.c.o ∂
    -sn STDIO=Main ∂
    -sn INTENV=Main ∂
    -sn %A5Init=Main ∂
    "{Libraries}"Stubs.o ∂
    "{CLibraries}"Runtime.o ∂
    "{CLibraries}"StdCLib.o ∂
    "{CLibraries}"CInterface.o ∂
    "{Libraries}"Interface.o ∂
    "{Libraries}"ToolLibs.o ∂
    -o MyProg
```

▶ Calling Traps in Pascal

The MPW Pascal compiler uses a set of interface files stored in a folder called PInterfaces, which is in the MPW Interfaces folder. The PInterfaces folder, like the CIncludes folder, contains an interface file for each Toolbox and operating system manager. However, to reduce the number of interface files that programs must access and to reduce memory requirements at compile time, a set of small files that provide indirect access to the most commonly used Toolbox interface files have been grouped together in a single file called ToolIntf. Similarly, a set of interface files that access commonly used operating system calls have been grouped together in an interface file called OSIntf.

Other interface files used in Pascal programs are Types, which provides the definitions of basic Pascal data types; QuickDraw, which provides an interface to QuickDraw; Traps, which contains the trap numbers of Toolbox and operating system traps; and Packages, which provides an interface to the Package Manager. Table 1-5 lists all the interface files that MPW Pascal uses.

Table 1-5. MPW Pascal interface files

ADSP.p	FileTransferTools.p
Aliases.p	FixMath.p
AppleEvents.p	Folders.p
AppleTalk.p	Fonts.p
Balloons.p	GestaltEqu.p
CommResources.p	Graf3D.p
Connections.p	HyperXCmd.p
ConnectionTools.p	IntEnv.p
Controls.p	Lists.p
CRMSerialDevices.p	MacPrint.p
CTBUtilities.p	Memory.p
CursorCtl.p	MemTypes.p
DatabaseAccess.p	Menus.p
DDEV.p	MIDI.p
Desk.p	Notification.p
DeskBus.p	ObjIntf.p
Devices.p	OSEvents.p
Dialogs.p	OSIntf.p
DisAsmLookup.p	OSUtils.p
DiskInit.p	Packages.p
Disks.p	PackIntf.p
Editions.p	PaletteMgr.p
EPPC.p	Palettes.p
ErrMgr.p	PasLibIntf.p
Errors.p	Perf.p
Events.p	Picker.p
Files.p	PickerIntf.p
FileTransfers.p	Power.p

Table 1-5. MPW Pascal interface files (continued)

PPCToolBox.p	Slots.p
Printing.p	Sound.p
PrintTraps.p	StandardFile.p
Processes.p	Start.p
QDOffscreen.p	Strings.p
Quickdraw.p	SysEqu.p
Resources.p	Terminals.p
Retrace.p	TerminalTools.p
ROMDefs.p	TextEdit.p
SANE.p	Timer.p
Scrap.p	ToolIntf.p
Script.p	ToolUtils.p
SCSI.p	Traps.p
SCSIIntf.p	Types.p
SegLoad.p	Video.p
Serial.p	VideoIntf.p
ShutDown.p	Windows.p
Signal.p	

In MPW Pascal, as in other versions of Pascal, interface libraries are accessed with a USES function. Because the MPW Pascal compiler uses streamlined interface files such as ToolIntf and OSIntf, the USES statement in an MPW Pascal program is usually much shorter than the series of #include statements that is required by a program written in MPW C.

In a simple Pascal application—one that makes calls to the Window Manager, the Menu Manager, the Dialog Manager, QuickDraw, and other commonly used managers—the USES statement that accesses interface files could be as simple as this:

```
USES
Types, QuickDraw, OSIntf, ToolIntf, Packages, Traps;
```

Calling CloseWindow in Pascal

In *Inside Macintosh*, this is the Pascal definition for the Window Manager call CloseWindow:

```
PROCEDURE CloseWindow(theWindow:WindowPtr);
```

Again, the WindowPtr argument in the call is a pointer to a WindowRecord, which is defined this way in MPW Pascal:

```
WindowRecord = RECORD
  port: GrafPort;
  windowKind: INTEGER;
  visible: BOOLEAN;
  hilited: BOOLEAN;
  goAwayFlag: BOOLEAN;
  spareFlag: BOOLEAN;
  strucRgn: RgnHandle;
  contRgn: RgnHandle;
  updateRgn: RgnHandle;
  windowDefProc: Handle;
  dataHandle: Handle;
  titleHandle: StringHandle;
  titleWidth: INTEGER;
  ControlList: ControlHandle;
  nextWindow: WindowPeek;
  windowPic: PicHandle;
  refCon: LongInt;
END;
```

This is how the CloseWindow call might look in a Pascal program:

```
CloseWindow(theWindow);
```

Starting up Tools in Pascal

In Pascal, as in C, most of the commonly used managers must be started up before they can be used in a program. Here is how managers are started in the MPW sample program TESample.p, which is written in Pascal:

```
InitGraf(@thePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs(NIL);
```

Compiling and Linking a Pascal Program

When you have finished writing a program in MPW Pascal, you must compile it and link it, just as you would compile and link a program written in MPW C. Again, you can create a makefile that can help you build your program by pulling down and selecting the Create BuildCommands item under the Build menu on the MPW menu bar, or you can write your own makefile. Rules for writing makefiles are covered in Chapter 8.

▶ Assembly Language Programming

In the prehistoric era of the personal computer era—that is, until about 1985 or so—most serious software for personal computers was written in assembly language. Today, times are changing; more and more applications for personal computers are being written in higher level languages such as C, Pascal, and C++.

If you want to write professional-quality software, however, it is still very useful to have some understanding of assembly language. When you compile and link a Pascal or C program, what you get is a program written in object code, or machine language. Also, when you debug an object-code program with a debugger such as MacsBug, the debugger disassembles the code into assembly language.

So, if you do not know anything about assembly language, there is no way that you can use MacsBug or any other object-code debugger. However, you don't have to know assembly language to use the source-level debugger SADE or other source-level debuggers that are available for the Macintosh.

Another good reason for learning as much as you can about assembly language is that there are some things you can do in assembly that you simply cannot do in a higher level language such as Pascal or C. For example, when you want to access a specific memory address or a specific microprocessor register, sometimes you may have to use assembly language to do it.

A knowledge of assembly language can also come in handy when you want to improve the way in which a Toolbox routine handles an operation. For example, the sample MPW program called TESample.c has a code segment written in assembly language that is linked to the main program by the MPW Linker after the main C program has been compiled. This assembly language segment, called a "glue" segment because of the way it is pasted into the program by the linker, is called TESampleGlue.a.

When the TESample.c program is run, TESampleGlue.a is called by the TextEdit routine TEClick when the mouse is clicked in a TextEdit control. TESampleGlue.a responds by calling a routine that implements automatic scrolling for a TextEdit field.

There are many other reasons why it is useful to have at least a basic understanding of assembly language. The most important reason is that you have to know something about assembly language in order to have a good understanding of Toolbox operations, the operating system, and other important components of a Macintosh computer system.

Calling Traps in Assembly Language

Inside the MPW Interfaces folder, there is a second folder called AIncludes. This AIncludes folder contains an interface file called Traps.a. The Traps.a file is a macro file that includes the A-line addresses of all the commonly used traps in the Macintosh Toolbox and operating system. For example, the address of the CloseWindow trap is listed as $A92D. All A-trap addresses are listed in *Inside Macintosh*.

The AIncludes folder also includes a number of equates files that are needed to assemble MPW assembly language programs. For example, the QuickEqu.a file contains QuickDraw equates, the ToolEqu.a file contains Toolbox equates, and the SysEqu.a file contains operating system equates.

Some Macintosh library routines are in library object files rather than in ROM. In *Inside Macintosh*, these routines are flagged with the notation "Not in ROM." To call the routines that these libraries contain, you must link your source code with the MPW file Interface.o. Then you must call the routines you need using assembly language JSR instructions.

To call a trap in assembly language, you must include the Traps.a file in your program with an INCLUDE statement. Other INCLUDE statements must usually be added so that the MPW assembler can find other equate files. In a typical assembly language program, these are some of the INCLUDE statements that would probably be included:

```
INCLUDE 'Traps.a'
INCLUDE 'ToolEqu.a'
INCLUDE 'PackMacs.a'
INCLUDE 'QuickEqu.a'
INCLUDE 'SysEqu.a'
```

When you have placed the necessary INCLUDE statements in an assembly language program, you can call any Toolbox or operating system trap using the appropriate trap macro in the opcode field of an assembly language instruction.

The names of all trap macros begin with the underscore character (_), followed by the name of the corresponding routine. For example, the macro for the Window Manager routine CloseWindow is _CloseWindow. So, to call CloseWindow, you would use an instruction with the macro name _CloseWindow in the opcode field.

Stack-Based Routines and Register-Based Routines

The calling conventions for Toolbox and operating system calls fall into two categories: stack-based routines and register-based routines. Stack-based routines pass their parameters via the stack, while register-based routines receive their parameters and return their results in 680X0 registers. As a rule, Toolbox routines are stack-based and operating system routines are register-based, but this is not always the case. In the entries listed for individual calls in *Inside Macintosh*, register-based calling conventions are supplied for all routines that use them; if none is shown, the routine is stack-based. This information is important because you have to set up parameters in the way that a routine expects before you can call it from any language.

Trap macros for Toolbox calls take no arguments, but those for operating system calls may have as many as three optional arguments. The first argument, if there is one, is used to load a register with a parameter value for the routine being called. The other arguments control the settings of the various flag bits in the trap word. The form of these arguments varies with the meanings of the flag bits and is described in *Inside Macintosh*, in the chapters on the relevant parts of the operating system.

Setting Up a Call's Parameters

To call a stack-based routine from assembly language, you must set up the call's parameters in the same way that the MPW Pascal compiler would if you were writing your program in Pascal. The numbers and types of parameters and the type of result returned by a function depend on the routine being called.

These are the steps you must use to make a trap call from assembly language:

1. If you are calling a function, reserve space on the stack for the result.
2. Push the routine's parameters onto the stack in the order in which they are listed in the routine's Pascal definition in *Inside Macintosh.*
3. Call the trap by executing the appropriate trap macro.

By the Way ▶

Getting Technical. When you call a trap, a return address is pushed onto the stack, along with an extra word of processor status information. Before the routine begins, the trap dispatcher removes this extra status word. The routine itself is responsible for removing its own parameters from the stack before returning. If it is a function, it will leave its result on top of the stack in the space reserved for it; if it is a procedure, it will restore the stack to the same state it was in before the call.

Calling CloseWindow in Assembly Language

For example, the CloseWindow function, as you have seen, is defined this way in Pascal:

```
PROCEDURE CloseWindow(theWindow:WindowPtr);
```

So here is how you would call CloseWindow from assembly language:

```
SUBQ.L    #4,SP              ; make room for result
MOVE.L    theWindow,-(SP)    ; push window pointer
_CloseWindow                 ; make the trap call
```

Starting up Managers in Assembly Language

It should come as no surprise to learn that in assembly language, as well as in C and Pascal, most managers must be started up before they can be used in a program. Here is a fragment of Sample.a, an assembly language program in the MPW Examples folder, in which some managers are initialized:

```
_InitGraf
_InitFonts
_InitWindows
_InitMenus
_TEInit
CLR.L   -(SP)
_InitDialogs
```

What Happens When You Call a Trap

When you issue an A-trap call, a circuit in the 680X0 processor called the 1010 trap emulator recognizes it as an unimplemented instruction (an instruction that begins with $A, or binary 1010) and generates a trap signal to the trap dispatcher. The trap dispatcher examines the bit pattern of the instruction to determine what operation it stands for, looks up the address of the corresponding routine in the trap dispatch table, and jumps to the routine.

The offset in a trap dispatch table entry is expressed in words instead of bytes, taking advantage of the fact that instructions must always fall on word boundaries, or even-byte addresses. These are the steps that the trap dispatcher goes through to find the absolute address of the routine:

1. It checks the high-order bit of the trap dispatch table entry to find out which base address to use.
2. It then doubles the offset to convert it from words to bytes (by left shifting one bit).
3. Finally, it adds the result to the designated base address.

As previously noted, a trap word always contains $A, or binary 1010, in bits 12 through 15. Bit 11 determines how the remainder of the word will be interpreted; usually it is 0 for operating system calls and 1 for Toolbox calls, although there are some exceptions.

Bits 0 through 9 of a trap word form the trap number—an index into the trap dispatch table—which identifies the routine being called.

By the Way ▶

A Bit of History. Bit 10 of a trap word, which some out-of-date books refer to as the "auto-pop bit," was originally reserved for use by language systems that could not generate inline trap calls, but instead did a JSR to the trap word, followed immediately by a return to the calling routine. The return address for the JSR was pushed onto the stack, followed by the return address. If the auto-pop bit was set, the trap dispatcher popped the trap's return address from the stack and returned directly to the calling program. The auto-pop bit is not used in modern development systems.

For operating system calls, only the low-order eight bits of the trap number (bits 0 through 7) are used. Thus, of the 512 entries in the trap dispatch table, only the first 256 can be used for operating system traps. Bits 8, 9, and 10 of an OS trap have specialized meanings that are covered in the assembly language chapter of *Inside Macintosh.*

▶ Making Toolbox Calls in C++

Using the Toolbox in a C++ program is very much like using it in a program written in MPW C. First you must include the interface files for the managers you will be using, like this:

```
#include <Types.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Controls.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <Scrap.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <Traps.h>
```

Next you must go through the standard C procedure for initializing Toolbox managers, which might look like this:

```
InitGraf((Ptr) &qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs((ResumeProcPtr) nil);
```

Then you can make all the calls you want to the Toolbox managers you have initialized and to those that initialize themselves at startup time.

▶ Making Toolbox Calls in MacApp

To use Toolbox and operating system managers in a MacApp program, you must include the necessary interface files in the program, just as you would in a program written in Pascal (or, if you are using MacApp with C++, a program written in C). Once you have included all the interface files you need, you can initialize the Toolbox managers that are used in your program by using the MacApp call InitToolbox, in this fashion:

```
InitToolBox;
```

That is all you have to do to get the Toolbox up and running in a MacApp program.

▶ Conclusion

This chapter focused on the features of the Macintosh that are most important to the Macintosh programmer: the User Interface, the Toolbox, and the operating system. It also provided an introductory explanation of how tools are called in MPW C, MPW Pascal, and MPW assembly language. More details on the Toolbox and the operating system and on writing Macintosh applications using MPW are presented in Chapters 5 through 8.

2 ▸ Commands and Scripts

MPW is not just an assembler or a compiler; it is a complete software development system, with more than 120 built-in commands that you can use to write, compile, link, and execute programs. If 120 commands are not enough for you, you can easily increase that number by writing commands of your own. You can customize the MPW environment in other ways, too. You can add items to the MPW menu, and you can write your own scripts, tools, and dialogs to carry out customized operations.

This chapter introduces some of the most important features of MPW and describes some of MPW's most important commands. It also tells—with the help of some hands-on programming examples—how to execute an MPW command, how to create a command of your own, and how to write an MPW script.

Other subjects covered in this chapter include:

- how to create aliases, or user-defined synonyms for command names
- how to use variables in scripts
- how to customize MPW operations by modifying the Startup and UserStartup scripts
- how to use the MPW online Help utility
- how to use file management commands

▶ The MPW Shell

MPW is built around a large application called the MPW shell. The shell includes both a text editor and a command interpreter. Various kinds of tools and external applications, including MPW's compilers and resource utilities, can be launched from within the shell.

Once you have your MPW system installed, you can launch the MPW shell by simply opening the MPW folder on your hard disk and double-clicking on the MPW application icon. You can also start MPW by double-clicking on any MPW document or tool icon.

▶ The MPW Worksheet Window

When MPW starts loading, a document called the MPW Worksheet window appears on your screen, as shown in Figure 2-1. MPW runs in a multi-window environment, so you can open other windows while the Worksheet window is open. In fact, you can display as many as 20 windows, probably more than you'll ever need to edit text and write programs.

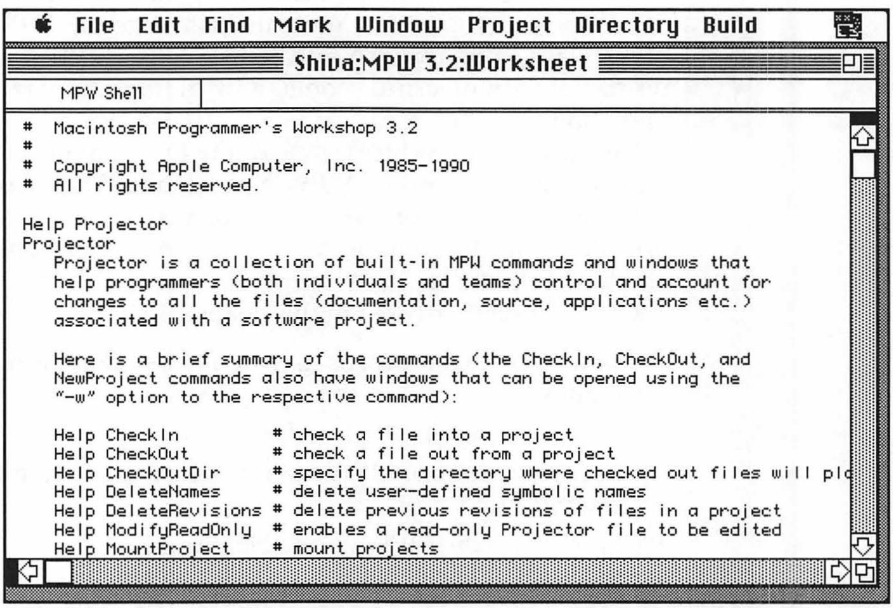


Figure 2-1. Worksheet window

At first glance, the MPW Worksheet window looks much like any other document window. But a closer inspection reveals some subtle but important differences.

▶ The Status Panel

One distinctive feature of the Worksheet window is a status panel in its upper left-hand corner, just below the title bar. (In earlier versions of MPW, the status panel was in the Worksheet window's lower left-hand corner, next to a slightly shortened horizontal scroll bar.)

When MPW is running a script (a series of commands), the shell uses the Worksheet window's status panel to display each MPW command as it is being executed. By watching the display panel while a script is running, you can monitor the shell's performance as it carries out each command. When no command is being executed, the words "MPW Shell" appear in the status panel.

▶ The Split-Window Feature

With the introduction of MPW 3.2, a split-window capability was added to the MPW Editor. With this feature, you can divide the Worksheet window (or any other MPW window) into scrollable panes, which you can use to view many different portions of a document simultaneously.

Take a close look at the top of the Worksheet's vertical scroll bar, and you'll see a small rectangle. There is a similar rectangle to the left of the window's horizontal scroll bar. These rectangles are called split bars.

If you press your mouse button inside the vertical split bar and drag the bar downward, the window splits into two horizontal panes, as shown in Figure 2-2. Both panes can now be scrolled independently, so you can use them to view two portions of the document in the window at the same time.

Now place your mouse button inside the horizontal split bar and drag it to the right. Another pane then opens, as shown in Figure 2-3. All three panes are independently scrollable, so you can now view three sections of the document in the window simultaneously.

You can open as many window panes as you like in this fashion, up to a maximum of 20. That's a lot of window panes, but if you have a big-screen Macintosh, you may one day need that many; who knows?

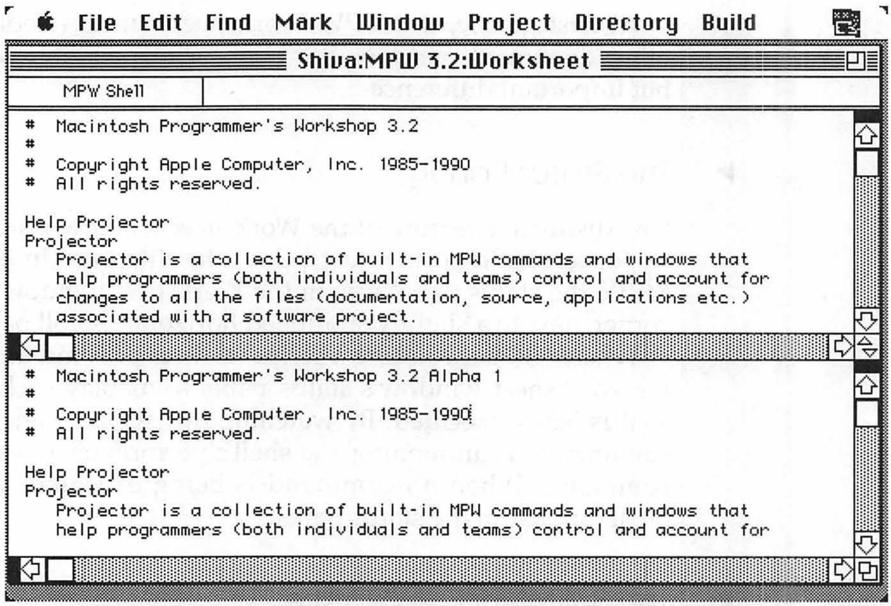


Figure 2-2. Horizontal split window

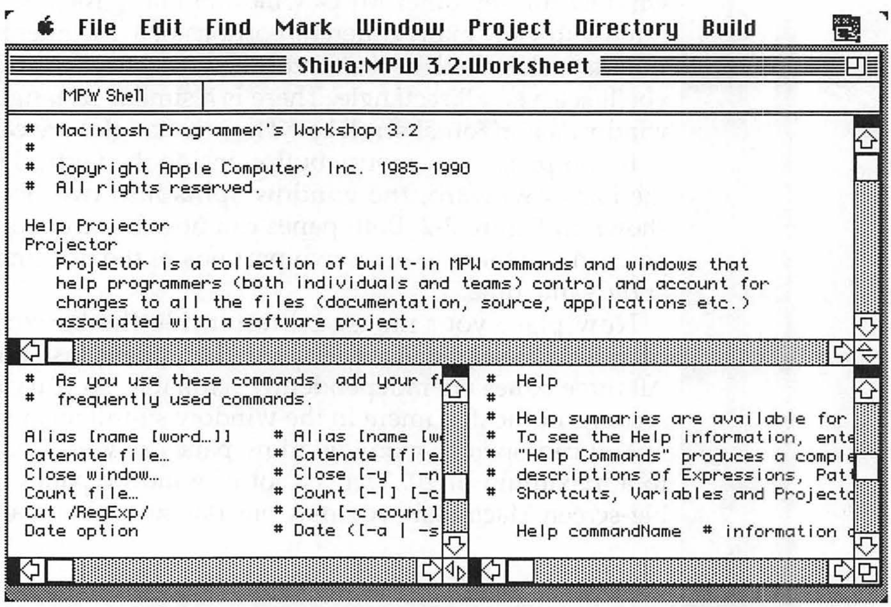


Figure 2-3. Window split three ways

▶ A Window That's Always Open

If you look at the Worksheet window very closely, you may notice that it has no Close box. That's because the Worksheet window is always present while MPW is running. Pull down the items under the MPW File menu, and you will see that you cannot close the Worksheet window by using any of them, either; the Close item is dimmed and disabled. In short, there is no way that you can get rid of the Worksheet window while MPW is running. You can place another window over it, but you cannot make it go away; it must always be there, because it is the command center for all MPW operations.

▶ The Worksheet Window's Title Bar

Another difference between the MPW Worksheet window and an ordinary document window is that the Worksheet window's full pathname is always displayed in its title bar. Since programmers often need to manipulate files and directories while using MPW, the full pathname of the Worksheet window can be a useful piece of information to have handy.

▶ The TileWindows and StackWindows Commands

When more than one window is open, there are two MPW commands—TileWindows and StackWindows—that can rearrange the windows on your screen. TileWindows reduces the sizes of all open windows and arranges them in a tiled pattern, so that you can see at least some of the contents of all of the windows on the screen.

The StackWindows command displays the active (topmost) window almost fullsize, and places all other open windows behind it in a stacked arrangement, so that you can see all their title bars.

Although the TileWindows and StackWindows commands can be issued from command lines, you can execute them more easily by selecting the Tile Windows and Stack Windows items from MPW's pull-down Window menu. The StackWindows and TileWindows commands are illustrated and described in more detail in Chapter 3, "Menus and Dialogs."

▶ The Browser Window

Another feature introduced in MPW 3.2 is the Browser window, shown in Figure 2-4.

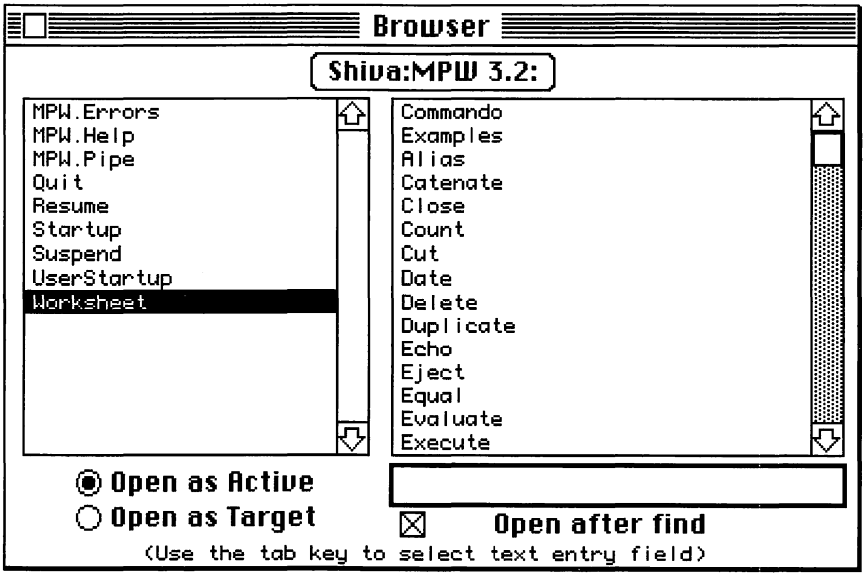


Figure 2-4. Browser window

You can display the Browser window by typing and entering the MPW command

```
Browser
```

It contains a list of files, a button to change directories, and a list of any markers that have been placed in the selected file (markers are described in Chapter 3). When you select a file shown in the Browser window, MPW opens it. When you select a marker, the shell opens the selected file, and finds and displays the marked selection.

The Browser window also includes a Text Edit field into which you can type a string that you want to find; a check box that determines whether the window that contains the string will be opened; and radio buttons that can be selected to determine whether a window opened by the Browser is to be the active window or the target window (active and target windows are described later in this chapter).

Note ▶

Clean Your Windows. The Worksheet window has one important feature that is not visible: When you exit MPW, any text that is in the Worksheet window is saved automatically, and MPW never clears anything from the Worksheet window unless you specifically ask it to. That means, of course, that the amount of text that MPW saves every time it closes the Worksheet window can grow quite large. So it is a good idea to take a look at your Worksheet from time to time and erase old work that you no longer need.

▶ The Target Window

In all but one respect, MPW follows Apple's User Interface Guidelines in the way that it handles windows. As the guidelines prescribe, MPW always treats the front window as the active window, and displays and highlights it in just the way that a well-behaved program should. And, also in accordance with the guidelines, all other windows are displayed and treated as inactive windows.

By the Way ▶

A Note About Window Management. The MPW package includes both a command-line interpreter and a full-screen, multi-window text editor. Generally speaking, when you want to issue an MPW command by typing and entering a command line, you will do that in the Worksheet window. But, when you want to compose or edit a text document—for example, the source code of a program—you will usually open a standard document window, and that is the window in which you will type your text and do your editing.

You don't have to manage your windows that way, of course; you could type documents in the Worksheet window and enter commands in a document window. But, since the Worksheet window is always open and since you can open and close document windows at will, the best approach is usually to enter commands in the Worksheet window and to compose and edit in a document window.

MPW makes one significant departure from the guidelines in the way it treats windows; it makes an important distinction between the second topmost window on the screen and any other inactive windows that may lie beneath it. In MPW, the second topmost window is displayed as an inactive window, but it is known as the target window and is treated differently from any other inactive window on the screen. The target window derives its name from the fact that it is the target of many shell commands; when you enter an editing command in an active window, the command is executed not in the active window, but in the target window.

The target window approach is used primarily in text editing. Suppose that you were working on a document in a document window and wanted to find and replace some text. You would not want to type a search-and-replace command into your document window because the command would wind up in the document—and, in this case, would actually edit itself. This is the kind of dilemma that the target window approach was designed to resolve.

By the Way ▶

How to Drag an Inactive Window. When you click your mouse anywhere in the structure region of a window, that window becomes the active window, and you can then drag it around with its title bar. Suppose you have a target window that you want to drag to another part of the screen, but you don't want it to become the active window. Just place your cursor in the target window's title bar and hold down the Command key on your keyboard as you press the button on your mouse. You can then drag your target window anywhere you like without making it the active window. This trick works not only on the MPW target window, but with any inactive window in any program.

▶ Searching for a String in the Target Window

Figures 2-5 and 2-6 show how the active window/target window approach works in MPW editing. In each illustration, the top window is the Worksheet window and the bottom window is a document window. You can tell from the way the windows are drawn on the screen that the Worksheet window is the active window and that the document window is the target window. The two figures illustrate a search for the string "charlie" in the target window.

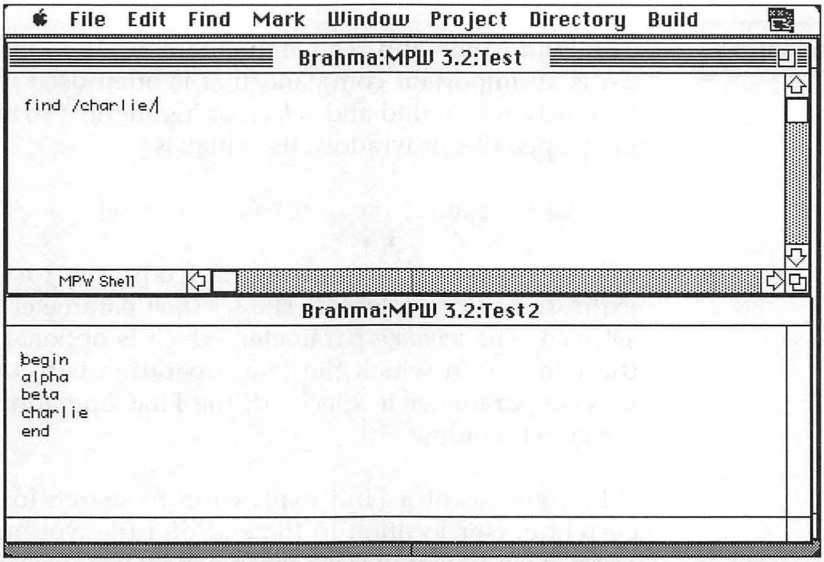


Figure 2-5. A 'Find' operation, part 1

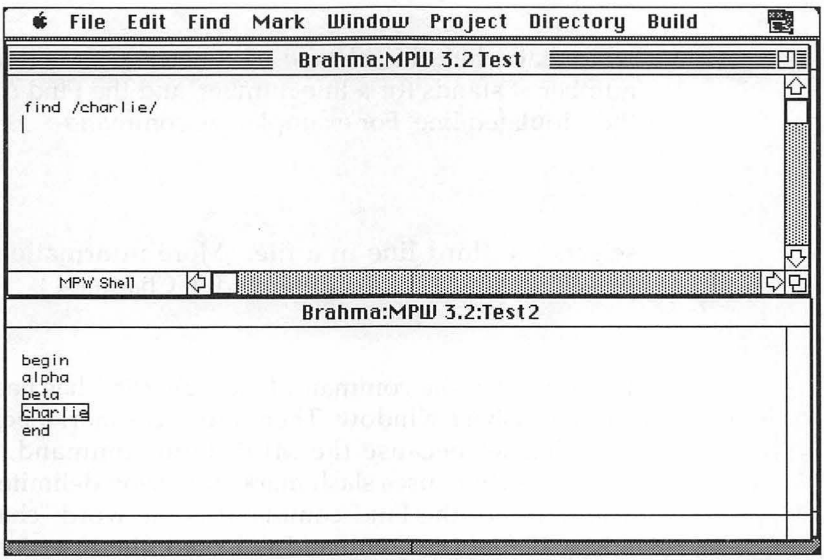


Figure 2-6. A 'Find' operation, part 2

Important ▶

The Find Command. The Find command used in Figures 2-5 and 2-6 is an important command that is often used in MPW editing. Its function is to find and select, or highlight, a string or expression in an open file, or window. Its syntax is

```
Find [-c count] selection [window]
```

The -c option is a number that equates to the number of expressions to be selected. The *selection* parameter is the text to be selected. The *window* parameter, which is optional, is the name of the window in which the Find operation is to take place. If no *window* parameter is specified, the Find operation takes place in the target window.

When you want a Find expression to search forward from the current cursor location to the end of a file, you must enclose the *selection* parameter in slash-mark delimiters (/.../). When you want the operation to proceed backwards toward the beginning of the file, you must place the *selection* parameter between backslashes (\...\).

Note that when the *selection* parameter of the Find command is a number, it stands for a line number, and the Find command selects the stipulated line. For example, the command

```
Find 3
```

selects the third line in a file. (More information about Find is presented later in this chapter and in Chapter 4.)

In Figure 2-5, the command Find /charlie/ has been typed on a line in the Worksheet window. There are slash marks before and after the word "charlie" because the MPW Find command, when going in a forward direction, uses slash marks for string delimiters.

The target of the Find command is the word "charlie" in the target window. Before the command is issued, the insertion point—the location of the text cursor—is placed at the beginning of the document in the target window.

The result of this Find operation is shown in Figure 2-6. The command Find /charlie/ has been typed, the Enter key has been pressed, and the search has been carried out. In the target document, the string "charlie" has been highlighted. The search has been successful.

This active window/target window approach to editing may seem unwieldy at first, but once you start working with MPW you will quickly get used to it. The system is actually quite intuitive, and you will probably be using it without even thinking about it by the time you finish the exercises in this chapter.

The MPW Command Language

The MPW package includes a powerful and versatile command language. With the more than 120 commands built into MPW, you can

- write, compile, link, and execute programs
- edit source code and other kinds of text
- control windows and other features of the Macintosh User Interface
- program the MPW shell
- manage files and directories

Table 2-1 lists the MPW command set. The same commands are listed in Appendix A, along with more details, such as syntax, options, parameters, and examples. MPW commands are listed according to their functions in Appendix B.

Table 2-1. The MPW command set

Command	*Function*
AddMenu	Add a menu item
Adjust	Adjust lines
Alert	Display an alert box
Alias	Define or write command aliases
Align	Align text to left margin
Asm	Assemble a program
AsmCvtIIGS	Convert APW Assembler source files to AsmIIGS format
AsmIIGS	Assemble an Apple IIGS program
AsmMatIIGS	Assembler source formatter
Backup	Folder file backup

Table 2-1. The MPW command set (continued)

Command	*Function*
Beep	Generate tones
Begin	Group commands
Break	Break from For-or Loop
Browser	Use the Browser tool to find files and selections
BuildCommands	Show build commands
BuildIndex	Create an index for a data file
BuildMenu	Create the Build menu
BuildMenuIIGS	Add CreateMakeIIGS to the Build menu
BuildProgram	Build the specified program
C	Compile a C program
Canon	Canonical spelling tool
Catenate	Concatenate files
CFront	C++ to C translator
CheckIn	Check a file into a project
CheckOut	Check a file out from a project
CheckOutDir	Specify the directory where checked-out files are to be placed
Choose	Choose or list network file server volumes and printers
CIIGS	Compile MPW IIGS C program
Clear	Clear the selection
Close	Close specified window(s)
Commando	Display a dialog interface for commands
Compare	Compare text files
CompareFiles	Compare text files and interactively view differences
CompareRevisions	Compare two revisions of a file in a project
Confirm	Display a confirmation dialog box
Continue	Continue with next iteration of For-or Loop
Copy	Copy selection to Clipboard
Count	Count lines and characters
CPlus	Script to compile C++ source
CreateMake	Create a simple makefile
CreateMakeIIGS	Create makefiles that build IIGS programs

Table 2-1. The MPW command set (continued)

Command	*Function*
Cut	Copy selection to Clipboard and delete it
Date	Write the date and time
Delete	Delete files and directories
DeleteMenu	Delete user-defined menus and menu items
DeleteNames	Delete user-defined symbolic names
DeleteRevisions	Delete previous revisions of files in a project
DeRez	Resource decompiler
DeRezIIGS	Resource decompiler for Apple IIGS
Directory	Set or write the default directory
DirectoryMenu	Create the Directory menu
DoIt	Highlight and execute a series of shell commands
DumpCode	Write formatted CODE resources
DumpFile	Display contents of any file
DumpObj	Write formatted object file
DumpObjIIGS	Dump OMF files
Duplicate	Duplicate files and directories
DuplicateIIGS	Copy files between Mac and GS/OS volumes
Echo	Echo parameters
Eject	Eject volume(s)
Entab	Convert runs of spaces to tabs
Equal	Compare files and directories
Erase	Initialize volume(s)
Evaluate	Evaluate an expression
Execute	Execute command file in the current scope
Exists	Confirm the existence of a file or directory
Exit	Exit from a command file
Export	Make variables available to commands
ExpressIIGS	Convert file(s) from OMF to ExpressLoad format
FileDiv	Divide a file into several smaller files
Files	List files and directories
Find	Find and select a text pattern
Flush	Flush tools that the Shell has cached
For	Repeat commands once per parameter

Table 2-1. The MPW command set (continued)

Command	*Function*
Format	Set or display formatting options for a window
Get	Get information about a keyword from a data file
GetErrorText	Display error message(s) based on message number
GetFileName	Display a Standard File dialog box
GetListItem	Display item(s) for selection in a dialog box
Help.MPW	Write summary information
If	Conditional command execution
Lib	Combine object files into a library file
Line	Find line in the target window
Link	Link an application, tool, or resource
LinkIIGS	The MPW IIGS Linker
Loop	Repeat commands until Break
Make	Build up-to-date version of a program
MakeErrorFile	Create error message textfile
MakeBinIIGS	Convert Load files to Binary files
MakeLibIIGS	Create IIGS Library files
Mark	Assign a marker to a selection
Markers	List markers
MatchIt	Semi-intelligent language-sensitive bracket matcher
MergeBranch	Merge a branch revision onto the trunk
ModifyReadOnly	Enable a read-only Projector file to be edited
Mount	Mount volume(s)
MountProject	Mount project(s)
Move	Move files and directories
MoveWindow	Move window (to horizontal, vertical location)
NameRevisions	Define a symbolic name
New	Open a new window
Newer	Compare modification dates of files
NewFolder	Create a new folder
NewProject	Create a new project
Open	Open file(s) in window(s)
OrphanFiles	Remove Projector information from a list of files
Parameters	Write parameters

Table 2-1. The MPW command set (continued)

Command	*Function*
Pascal	Compile Pascal program
PascalIIGS	The MPW IIGS Pascal Compiler
PasMat	Pascal programs formatter
PasRef	Pascal cross-referencer
Paste	Replace selection with Clipboard contents
PerformReport	Generate a performance report
Position	Display current line position
Print	Print text files
ProcNames	Display Pascal procedure and function names
Project	Set or write the current project
ProjectInfo	Display information about a Project
Quit	Quit MPW
Quote	Echo parameters, quoting if needed
Rename	Rename files and directories
Replace	Replace the selection
Request	Request text from a dialog box
ResEqual	Compare resources in two files
ResEqualIIGS	Compare resources in two Apple IIGS files
Revert	Revert window to previous saved state
Rez	Resource compiler
RezDet	Detect inconsistencies in resources
RezIIGS	Resource compiler for Apple IIGS
RotateWindows	Send active (frontmost) window to back
Save	Save specified windows
SaveOnClose	Save window when it closes
Search	Search files for pattern
Set	Define or write shell variables
SetDirectory	Set the default directory
SetFile	Set file attributes
SetPrivilege	Set access privileges for directories on file servers
SetVersion	Maintain version and revision number
Shift	Renumber command file positional parameters
ShowSelection	Scroll window, setting selection to desired position

Table 2-1. The MPW command set (continued)

Command	*Function*
Shutdown	Power down or restart the machine
SizeWindow	Set a window's size
Sort	Sort or merge lines of text
StackWindows	Arrange windows with title bars showing
StreamEdit	Non-interactive, script-driven editor
Target	Make a window the target window
TileWindows	Arrange windows in a tiled fashion
TransferCkid	Move Projector information from one file to another
Translate	Translate characters
Unalias	Remove aliases
Undo	Undo the last edit
Unexport	Remove variable definitions from the export list
Unmark	Remove a marker from a window
Unmount	Unmount volume(s)
UnmountProject	Unmount project(s)
Unset	Remove shell variable definitions
UserVariables	Set all user variables (uses Commando)
Volumes	List mounted volumes
WhereIs	Find the location of a file
Which	Determine which file the shell is to execute
Windows	List windows
ZoomWindow	Enlarge or reduce a window's size

▶ Four Varieties of MPW Commands

When you categorize commands by the way they are written, MPW has four kinds of commands: built-in commands, scripts, tools, and applications.

Built-in Commands

Built-in commands are commands that are part of the MPW shell. The familiar editing commands Cut, Paste, and Copy are examples of built-in commands.

Scripts

Scripts are text files made up of commands. In MPW, you can combine any sequence of commands into a text file and then execute the entire file by simply typing and entering its name as a command. Since scripts are made up of commands, they are sometimes referred to as command files.

A special script called the Startup script is executed every time MPW is launched. It defines a list of default variables and alias definitions (alternate names for commands) that are recognized by all other scripts. It is therefore known as a command script. After MPW executes the Startup script, it also runs another command script called the UserStartup script. You can customize MPW by adding lines to the UserStartup script or by editing lines that it already contains. By modifying the UserStartup script, you can add customized variables and aliases to MPW. In fact, you can even add your own menus and menu items to the MPW menu bar. Ways in which you can edit the UserStartup script are described later in this chapter and in Chapter 3.

MPW Tools

MPW Tools such as C (for compiling C programs) and Link (for linking files) are executable programs that are stored as files on a disk and are completely integrated with the shell environment. An MPW tool can be either an MPW script or an executable program.

Applications

Applications are standard application programs such as ResEdit, MacPaint, programs you have written, or any software package that you can buy. Applications may not know about MPW, but it knows about them; when MPW is running, you can execute any application from the shell environment by simply typing and entering its name as a command.

▶ The Structure of a Command

In MPW, a command is defined as a list of words separated by blanks (either spaces or tabs) and ending with a command terminator.

The first word of a command is always the command name. The name of the command can stand alone, or it can be followed by options, parameters, or both. A command is always terminated by a command terminator. The general form of an MPW command is

```
commandName [options] [parameters...] commandTerminator
```

The Command Name

The name of a command is either the name of a built-in MPW command or the file name of a program, script, or tool to execute. Command names are not case sensitive. Alternative names, called aliases, can be defined for the names of commands. When you have defined an alias for a command, you can use it in commands and scripts in the same way you would use the actual name of the command. Procedures for defining aliases are presented later in this chapter.

By the Way ▶

A Command Decision. In most computer languages, a reserved word used to carry out an operation is usually referred to as a command, and a line that is executed to carry out an operation is usually called a statement. In the *MPW 3.0 Reference,* the term "statement" is not used, and no clear distinction is made between commands and statements. A command word is sometimes called a command and is sometimes called a command name. With reluctance, but for the sake of consistency with the *MPW 3.0 Reference,* I have decided to be just as vague about commands and statements as the *MPW 3.0 Reference* is. Therefore, in this book, the word "command" sometimes refers to a statement, and other times it refers to a command. The intended meaning of the word should always be clear from the context.

Options and Parameters

In an MPW command line, options are letters preceded by a negative sign, for example, -t. As their name implies, they are optional. When an option is included in a command line, it alters the operation that the command performs.

A command can also be followed by one or more parameters. In most commands, parameters contain information that is passed on to the command being issued. But the shell interprets certain parameters—such as those used in I/O redirection—before the command is executed, and thus they are not passed on to the command itself. The shell also expands any variables that a command may contain before it executes the command. More information on I/O redirection and variables is presented later in this chapter.

Note ▶

A Note About Options and Parameters. In the *MPW 3.0 Reference,* command-line options—letters preceded by a negative sign (such as -t)—are referred to as parameters. Real parameters—full words not preceded by a negative sign—are called parameters, too.

In my opinion, options and parameters are different. Options tell the command interpreter how an operation should be carried out, whereas what I call parameters are the targets, or objects, of commands. Since options differ from parameters in function as well as form, this book places options and parameters in different categories. Although this is not consistent with the style of the *MPW 3.0 Reference,* it should not cause you any confusion.

Command Terminators

Every MPW command is terminated by a command terminator. The most commonly used command terminator is the Return character. A Return character always ends a command unless it is preceded by a line-continuation character ∂, as explained in the next section.

▶ Multiple-line Commands

You can continue a command on the next line by typing ∂ (Option-D) followed by a Return. When the shell interprets the command, it discards both the ∂ character and the Return before it executes the command. To use the ∂ character as a line-continuation character, you must type a Return character immediately after the ∂, with no blanks or comments separating them.

To see how the line-continuation character ∂ is used, type

```
Echo This sentence appears ∂
all on one line.
```

Then execute the command by selecting (highlighting) both lines and pressing the Enter key. The Echo command should then write

```
This sentence appears all on one line.
```

to your screen.

The ∂ character also has another function: It is used as an escape character to insert certain nonprinting characters into text. When ∂ is followed immediately by the letter *n*, it inserts a newline character, or a Return, into a document. When it is followed by the letter *t*, it inserts a Tab. When it is followed by the letter *f*, it inserts a form feed. For example, the command

```
Echo ∂n
```

prints a newline character on the screen, just as if the user had typed a Return.

You can also prevent MPW from interpreting a special character by preceding the character with ∂. For example, if you try to execute the command

```
Echo *
```

MPW responds with the following error message.

```
### MPW Shell - File name pattern "*" is incorrect.
```

But if you issue the command

```
Echo ∂*
```

MPW prints

```
*
```

to your screen. If you enclose the ∂ character in quotation marks, the shell does not recognize it as a special character, but treats it like any other typed character. Table 2-2 shows the ways in which ∂ can be used.

A command can also be terminated by a semicolon (;), a pipe symbol (|), or a conditional execution operator. Each of these special characters can, in turn, be followed by a Return.

Table 2-2. ∂ character uses

Characters Typed	*Result*
∂	Alone at the end of a line; line-continuation character
∂n	Return character (ASCII CR)
∂t	Tab character (ASCII HT)
∂f	Form feed (ASCII FF)
∂-Return	Both ∂ and return character ignored
∂*char*	Display *char* (if not *n*, *t*, *f*, or Return)

By using a semicolon as a command terminator, you can type more than one command on a line. For example, the three command lines

```
Beep
Beep
Beep
```

and the command line

```
Beep ; Beep ; Beep
```

do the same thing; they execute three Beep commands, causing the Macintosh speaker to beep three times.

The special-character combinations && and || are logical operators as well as command terminators. If you separate two commands with the characters &&, the second command is executed only if the first command succeeds. Conversely, if you separate two commands with the characters ||, the second command is executed only if the first command fails. For example, the line

```
Find /charlie/ && Echo Found!
```

searches for the string "charlie" in a file and echoes the exclamation "Found!" if the string is found. The line

```
Find /zebra/ || Echo Sorry!
```

echoes the message "Sorry!" if the Find command fails.

Important ▶

The Echo Command. Echo—short for "Echo parameters"—is the command that is most often used to pass literal strings to MPW commands. You can use Echo to monitor the operation of scripts while they are running, and to check the results of variable substitution, command substitution, and file name generation, which are covered later in this chapter. The syntax of the Echo command is:

```
Echo [-n] [parameter...] [> parameters...]
```

Echo writes its parameters, separated by spaces and terminated by a Return, to MPW's standard output, normally the active window. If no parameters are specified, Echo writes only a Return.

When the -n option is used with the Echo command, it means, "Don't write a Return following the parameters." That means that the insertion point—the location of the text cursor—remains at the end of the output line after the text to be echoed is written. The -n option itself is not echoed.

When the | character is used between two commands, it passes, or pipes, the output of the first command to the input of the second. For example, the line

```
Files | Count -l
```

pipes a list of files to the Count command. Count then counts the number of files on the list and echoes its results to standard output, in this case the screen. The Files command is described in more detail later in the chapter.

Important ▶

The Count Command. You can use the Count command to count the number of lines or characters in a file. Its syntax is:

```
Count [-l] [-c] [file...]
```

If you use the -l option with the Count command, it counts the number of lines in the specified file. If you use the -c option, it counts the characters in the file. By using redirection operators, described later in this chapter, you can direct the output of Count to a specified window or file.

Another good example of piping is a command that assembles and links a source file and reports on whether the operation succeeded. In the following example, the Asm command is used to assemble an assembly language source file, and the Link command is used to link it.

```
Asm Sample.a && Link Sample.a.o -o ∂
Sample.Code || (Echo Failed; Beep)
```

If the assembly succeeds, the command links the object file that is generated by the Asm command. But if either the assembly or the link operation fails, the command echoes the message "Failed," and beeps a warning. The Asm and Link commands are described in Chapter 8.

In the first line of the preceding example, the ∂ character (Option-D) is used in its line-continuation role; it causes MPW to treat both lines of the command as if they were on a single line. In the second line, parentheses group the Echo Failed and Beep commands together so that they are executed as a unit if the assembly fails.

MPW's command terminators are listed in Table 2-3.

Table 2-3. Command terminators

Terminator	*Example*	*Description*
Return	cmd1(r) cmd2	Ends cmd1 and moves to the next line.
;	cmd1 ; cmd2	Executes cmd1 and then executes cmd2, allowing more than one command to appear on a single line.
&&	cmd1 && cmd2	Executes cmd2 only if cmd1 succeeds (that is, returns a status code of 0).
||	cmd1 || cmd2	Executes cmd2 only if cmd1 fails (that is, returns a nonzero status code).
|	cmd1 | cmd2	Pipes the output of cmd1 to cmd2.

▶ Command and Parameter Syntax

If a parameter contains more than one word, it is usually necessary to enclose the parameter in quotation marks so that MPW recognizes it as a single parameter. However, there are exceptions to this rule. For example, you can pass a multi-word parameter to the Echo command without enclosing it in quotes. Thus the commands

```
Echo This is a string
```

and

```
Echo "This is a string"
```

have identical results; they both write

```
This is a string
```

to standard output, normally the screen.

The reason that quotes are not needed around a parameter to the Echo command is that the Echo command echoes back all the parameters passed to it, separated by spaces. (You can also use the -n option with Echo if you do not want a Return to be echoed, but that does not confuse MPW either, since options in a command always begin with minus signs).

If a parameter consists of only one word, quotes may be used, but they are not necessary unless the word contains a special character or a variable that contains blanks or special characters. More details about special characters and variables are presented later in this chapter.

Differences Between Single and Double Quotes

You can use either single or double quotation marks to delimit a parameter, but MPW treats single quotes and double quotes differently. When you use double quotes in a command, the shell does some work on the command before it executes the command. Specifically, the shell interprets the ∂ character (Option-D), and expands variables and defines aliases, before it carries out the command. When you enclose a command in single quotes, everything inside the quotes is taken literally. Examples of what these differences mean in MPW commands are presented later in this chapter and in Chapter 3.

Important ▶

The Comment Character #. MPW's comment prefix is the # symbol. When you place a comment on a command line, MPW ignores all text from the # symbol to the next command terminator.

You can place the comment character at the beginning of the line, or anywhere thereafter. A comment placed at the end of a command looks like this:

```
Echo Print this            # but don't print this.
```

In an MPW script, you can "comment out" a line—that is, prevent it from executing—by typing the # symbol in front of the line. For example, if you had a script that executed two Beep commands, but you wanted to eliminate the second beep temporarily, you could comment out the second beep like this:

```
Beep
# Beep
```

Later, if you wanted to put the second beep back into your script, you could remove the # symbol and restore the second Beep command.

If a command line ends with the line-continuation character ∂, the ∂ character has no effect on comments; they still end at the physical end of the line. For example, if you execute the command lines

```
Echo How                 # a comment ∂
are you this morning     # another comment
```

MPW prints the line

```
How are you this morning
```

on the screen.

Using Quotes Within Quotes

When you use quotation marks in a command, MPW expects them to be used in pairs. Hence, if you try to execute a command that contains only one quotation character, like this:

```
Echo "This is a string      # This is incorrect
```

MPW returns this error message:

```
### MPW Shell - "s must occur in pairs.
```

But if you use quotes in pairs, like this,

```
Echo "This is a string"
```

there is no error.

If a multi-word parameter contains an apostrophe—which MPW interprets as a single quotation mark—you can prevent the parameter from causing an error by enclosing the parameter in double quotation marks like this:

```
Echo "There's a great day coming"
```

Conversely, a parameter that includes double quotation marks can be enclosed in single quotes, as long as it does not contain any variables, aliases, or ∂ characters:

```
Echo 'This program is called "Bananas.c."'
```

If you want to use both single and double quotation marks in a parameter, you can use both kinds of quotes in a nested fashion. For example, the command

```
Echo '"I '"won't"' do it," she said.'
```

echoes this message:

```
"I won't do it," she said.
```

To familiarize yourself with the way in which MPW interprets quotation marks, you might find it helpful to experiment with some of your own examples.

How MPW Interprets Commands

MPW goes through seven steps when it interprets a command:

1. Alias substitution
2. Evaluation of structured constructs
3. Variable and command substitution

4. Blank interpretation
5. File name generation
6. I/O redirection
7. Evaluation and execution

Alias Substitution

Aliases are alternate names for MPW commands. MPW defines some aliases when it starts up, and you can also define your own. When MPW starts interpreting a command, the first thing it does is scan the command for any words that are aliases. When an alias is found, it is interpreted or translated back into the actual name of its corresponding command.

Evaluation of Structured Constructs

MPW commands may be simple or structured. A simple command consists of a single keyword, either standing alone or followed by any combination of options, parameters, or both. Structured commands are commands that let you control the order in which other commands are executed.

Structured constructs are programming constructs that perform conditional execution and looping operations. The commands Begin, For, If, and Loop are used to begin structured constructs and are therefore known as structured statement openers. Every structured statement opener must stand alone on a line, and each must be followed by an End command that stands at the beginning of a subsequent line. Structured constructs used in MPW include the constructions Begin . . . End, If . . . Else . . . End, For . . . End, and Loop . . . End.

Since a structured construct is made up of more than one command, structured constructs are used primarily in scripts.

Commands that can be used in structured constructs are listed in Table 2-4. Examples of how structured constructs are used in MPW scripts are presented at the end of this chapter.

Table 2-4. Commands used in structured constructs

Command	*Usage*
Begin...End	Enclose a conditional structure.
If...	Perform a conditional execution.
If...Else	Perform a conditional execution with optional Else . . . .
If...Else...Else If...	Perform a conditional execution with optional Else . . . and Else If . . . .
For...	Repeat commands once per parameter.
Loop	Repeat commands until Break.
Break	Break from For or Loop.
Break If...	Break with optional If . . . .
Evaluate	Evaluate an expression.
Execute	Execute a script in the current scope.
Exit	Exit from a script.
Continue	Continue with next iteration of For or Loop.

Variable Expansion and Command Substitution

In MPW, variables are defined with the Set command. Once a variable has been defined, its name must be enclosed in curly braces ({}) when it is used in a command.

Some variables are defined by the MPW shell, and others—such as variables that equate to pathnames—are predefined when MPW is launched and are automatically redefined if the values that they equate to are changed. You can also define your own variables, as explained later in this chapter.

In MPW, variables are not typed; all variables equate to strings of text. During the variable expansion stage of command interpretation, the shell also expands any variables that are delimited by slash bars (/.../), backslashes (\...\), or *double* quotation marks ("..."). However, if a variable is enclosed in *single* quotation marks ('...') like this:

```
'{MPW}'
```

it is not expanded.

The {MPW} variable, as explained later in this chapter, is a shell variable that equates to the current pathname of the MPW folder. Since variables delimited by double quotation marks are translated into their actual values during the variable expansion process, the command

```
Echo "{MPW}"
```

echoes the *contents* of the {MPW} variable, in this fashion:

```
HD:MPW:
```

Because variables enclosed in single quotation marks are not expanded, the command

```
Echo '{MPW}'
```

echoes the string

```
{MPW}
```

which is a very different result!

During variable expansion, the shell also checks to see whether or not the ellipsis character (Option-;, or ...) has been used in a command. If an ellipsis character is found, the shell executes the Commando command, which displays a Commando dialog. The output of the Commando dialog then replaces the command being interpreted. Commando dialogs and the Commando command are covered in Chapter 3.

Table 2-5 lists the kinds of variables used in MPW. More information about variables is presented later in this chapter.

Table 2-5. Kinds of variables used in MPW

Kind of Variable	*Examples*	*Description*
Shell	{MPW}, {Boot}	Includes predefined variables and startup variables.
Predefined	{Boot}	Used for pathnames; set by the shell and maintained automatically.
Startup	{MPW}	Used for pathnames, and for display and printing defaults; can be redefined by modifying the Startup script.

Table 2-5. Kinds of variables used in MPW (continued)

Kind of Variable	*Examples*	*Description*
User	{TileOptions}	Initially defined by MPW, but can be redefined in the Startup or UserStartup script.
User-defined	{Fred}	Defined by user; can be defined in a command, in a user-created script, or in the Startup or UserStartup script.
Parameter	{0}, {#}, {1}, {2}...{n}, {Parameters}, {"Parameters"}	Variables that equate to parameters in scripts.

Command substitution, which MPW performs along with variable expansion, is the process of using one command to get the parameters of another. In a command line or a script, you can instruct MPW to perform command substitution by delimiting a sequence of one or more commands with the backquote character (`). When one or more commands are set off in this way, MPW carries out each command and then passes the results back to a previous command on the same line—in much the same way that a language such as C returns the result of a function. For example, if you execute the command

```
Duplicate `Files -t TEXT` "{Boot}InsideMPW"
```

the Files command finds all files of type TEXT in the current directory and builds a list of file names. Files then passes the list to the Duplicate command, which copies each file into a directory on the Boot disk named InsideMPW.

In this example, "{Boot}" is a shell variable that is always defined as the Boot disk. There's more about the Files command, and about {Boot} and other shell variables, later in this chapter. Examples illustrating the use of the command substitution delimiter (`) are presented later in this chapter and in Chapter 4.

Blank Interpretation

As mentioned eariler in this chapter, blank spaces are used in MPW commands to separate command names, options, and parameters. Blank spaces can also appear in file names, but if they do, they must be enclosed in quotation marks so that MPW does not mistake them for the blanks that separate words in commands. The rules for using single and double quotation marks in file names are the same as those for using single and double quotes in strings and expressions.

As pointed out earlier, MPW defines a blank as an unquoted space or a tab. During the blank-interpretation stage of command interpretation, MPW divides the text of a command into individual words separated by blanks.

Some special characters, called operators, are always considered separate words, whether or not they are separated from other words by blanks. That's a convenience for you; it means that when you use these characters, you can be a little sloppy about spacing and MPW won't care. Operators that do not have to be surrounded by spaces are:

```
;  |  ||  &&  (  )  <  >  >>  ≥  ≥≥  …  Σ  ΣΣ
```

These are not the only operators used in the MPW command language, but they are the only ones that *never* have to be surrounded by spaces in MPW commands.

Within expressions that follow the structured statement opener If, and in expressions that follow the Evaluate command, *all* operators that are not enclosed in quotation marks are considered separate words. There's more about redirection operators, structured constructs, and the Evaluate command later in this chapter.

Operators that never have to be surrounded by spaces are listed in Table 2-6.

Table 2-6. Operators used in MPW commands

Operator	*Type*	*Function*
;	Command terminator	Separates multiple commands on the same line.
|	Command terminator	Separates commands and pipes output to input.
||	Command terminator	Separates commands, executing second if first fails.

Table 2-6. Operators used in MPW commands (continued)

Operator	*Type*	*Function*
&&	Command terminator	Separates commands, executing second if first succeeds.
(	Delimiter	Groups characters or commands together.
)	Delimiter	Groups characters or commands together.
<	Redirection operator	Takes command input from file name parameter.
>	Redirection operator	Sends command output to file name parameter.
>>	Redirection operator	Appends command output to file name parameter.
≥	Redirection operator	Sends diagnostic output to file name parameter.
≥≥	Redirection operator	Appends diagnostic output to file name parameter.
Σ	Redirection operator	Sends both standard and diagnostic output to file name parameter.
ΣΣ	Redirection operator	Appends both standard and diagnostic output to file name parameter.
…	Commando operator	Displays a Commando dialog and substitutes output of dialog for output of command.

File Name Generation

The MPW command language has eight special characters that can be used to perform special functions in file names. They are:

```
?  ≈  [ ]  *  +  « »
```

These eight characters can also be used as operators in regular expressions. If they appear in expressions that are delimited by single or double quotes, or by the slash delimiters / or \, MPW interprets them as regular expression operators. If they are not quoted, MPW interprets them as file name generation operators.

File name generation operators have the same meanings in MPW file names that they have when they are used as regular expression operators in quoted expressions. The characters ? and ≈ are wildcard characters; the characters [and] are brackets that enclose patterns; the characters * and + stand for repetitions of a specified character; and the characters « and » enclose numbers that specify the number of times an operation is to be performed. MPW's file name generation operators and their functions are listed in Table 2-7.

Table 2-7. File name generation operators

Symbol	*Description*	*Function*
?	Question mark	Matches any single character (except a colon, which cannot be used in a file name).
≈	Option-X	Matches any string of zero or more characters (except a colon).
[*characterList*]	Character list in braces	Matches any character in the list.
[¬*characterList*]	Character list in braces, preceded by Option-L	Matches any character not in the list.
*	Star	Zero or more repetitions of the preceding character or character list.
?*	Question mark and star	Same as ≈.
«*n*»	Number in European quotes (Option-\ and Option-Shift-\)	Specifies the number of repetitions of the preceding character or character list.
+	Plus sign	One or more repetitions of the preceding character or character list.

In addition to being used as file name generation operators, the characters listed in Table 2-7 can also be used as operators in regular expressions. If these characters are delimited by single or double quotes, or by the slash delimiters / or \, MPW interprets them as regular expression operators. If they are not quoted, MPW interprets them as file name generation operators.

If an unquoted word in a command contains a file name generation operator, it is considered a file name pattern. When a file name pattern is encountered in a command, MPW replaces the pattern with an alphabetically sorted list of file names that the pattern matches. Then, if the command you are using is one that lists file names—such as Files or Volumes—the list that has been generated is written to standard output.

For example, the command

```
Files ≈
```

lists all the files in the current directory. The command

```
Files ≈.p
```

lists all the files in the current directory with names that end with the extension ".p". The command

```
Files Source.?
```

lists every file in the current directory whose name begins with the name "Source," followed by a single character, for example, Source.c, Source.p, Source.a, and Source.r. And the command

```
Files Source≈
```

lists each file in the current directory with a name that begins with the word "Source," followed by any number of characters: for example, Source, Source.c, Source.p, Source.a, Source.r, Sourcerer and SourceFile.

For more information about regular expression operators and file name generation operators, see Chapter 4.

I/O Redirection

By default, MPW provides all built-in commands, scripts, and tools with three open files: standard input, standard output, and diagnostic output. Standard input comes from the console (the window where the command is executed); standard output and diagnostic output are returned to the console immediately following the command. (How you can override these default assignments with the symbols <, >, >>, ≥, ≥≥, Σ and ΣΣ is explained later in this chapter.)

Evaluation and Execution

The last step in the interpretation of a command is evaluation and execution. First, MPW evaluates the command to determine whether any errors have been encountered. If there are no errors, the command is executed. If an error is detected, an error message is returned.

If the command-line interpreter encounters an error while executing a command, it returns a negative status code. If no error is encountered, it returns a status code of 0. The status code returned by the command-line interpreter is returned in the shell variable {Status} (which is described in more detail later in this chapter).

Table 2-8 lists all the negative status codes, or error codes, that the command-line interpreter can return. MPW returns errors in a definable diagnostic output, which is explained later in this chapter.

Table 2-8. Command-line error codes

Error	*Meaning*
-1	Command not found, script is a directory, script is not executable, or script has a bad date.
-2	File name expansion failed, or there was an error in the expression syntax.
-3	Bad syntax: error in the control constructs, quotation characters and braces were not balanced, or command was missing end or ")" characters.
-4	Missing file name following I/O redirection, or the file could not be opened.
-5	Invalid expression (used with commands such as If, Break If, Continue If, or other such constructs).
-6	Tool could not be started.
-7	Runtime error during tool execution, most likely an out-of-memory error.
-8	User aborted the tool from the debugger.
-9	User aborted the tool with Command-period.

Tips for Writing Command Lines

Before you start executing commands using the MPW command language, consider the following tips and shortcuts for writing command lines:

- You can type an MPW command line in the same way that you would type ordinary text. To send the command to MPW's command interpreter for execution, press the Enter key on your numeric keypad—not the Return key on the main part of your keyboard.
- The cursor does not have to be at the end of a command line when you press Enter. You can place the cursor anywhere over the line and press Enter.
- If you do not want to use the Enter key, you can press Command-Return instead, but you will probably find that more cumbersome. Another alternative is to click your mouse inside the status panel in the upper left-hand corner of the MPW window.
- You can also enter an MPW command by dragging the mouse to select a range of text, and then pressing Enter. Using this method, you can select as many command lines as you like, and MPW executes them all.
- You can select a full line of text by triple-clicking the mouse anywhere on the line. If you triple-click on a command line and then press Enter, MPW executes the command.
- You can save a series of commands as a command file, or script. Then you can execute your entire script by typing its name as an MPW command. (You'll get an opportunity to write and execute some scripts later in this chapter.)
- When you are working with a document in a window, you can use the Command (Apple) key to move quickly through blocks of text. To move to the left or right margin, press Command-Left Arrow or Command-Right Arrow. Command-Shift-Up Arrow takes you to the top of the document, and Command-Shift-Down Arrow takes you to the bottom.
- The MPW command interpreter is not case-sensitive (although there are some default case-sensitivity settings you can change), so you can type the commands presented in this chapter in uppercase letters, lowercase letters, or a combination of both. For now, as far as MPW is concerned, they are all the same.

Note ▶

Sensitive Cases. In case you're curious about what is case sensitive in MPW and what isn't, here is the answer.

- By default, MPW is not case-sensitive.
- File names, aliases, variables, and commands are *never* case-sensitive.
- You can change the case-sensitivity of everything else by setting the variable {CaseSensitive}, as explained later in this chapter.

Table 2-9 lists some shortcuts and other tips that you might find handy when you use MPW.

Table 2-9. Shortcuts for using MPW

Moving Selection Points and Deleting Text

Shortcut	*Effect*
Up Arrow	Moves selection point one line above current selection.
Down Arrow	Moves selection point one line below current selection.
Right Arrow	Moves selection point one character to the right.
Left Arrow	Moves selection point one character to the left.
Command-Shift-Up Arrow	Moves selection point to top of file.
Command-Shift-Down Arrow	Moves selection point to bottom of file.
Command-Down Arrow	Moves selection point down one screen size.
Command-Right Arrow	Moves selection point to right edge of current line.
Command-Up Arrow	Moves selection point up one screen size.
Command-Left Arrow	Moves selection point to left edge of current line.
Command-Backspace	Deletes text from current selection to end of file.
Shift-Left Arrow	Extends selection to the left by one character.

Table 2-9. Shortcuts for using MPW (continued)

Moving Selection Points and Deleting Text (continued)

Shortcut	*Effect*
Shift-Right Arrow	Extends selection to the right by one character.
Shift-Up Arrow	Extends selection upward one line.
Shift-Down Arrow	Extends selection downward one line.
Option-Shift-Left Arrow	Extends selection to the left by one word.
Option-Shift-Right Arrow	Extends selection to the right by one word.

In Dialogs Without a TextEdit Item

Action	*Meaning*
Type "Y"	Yes
Type "N"	No
Command-Period	Cancel
Escape key	Cancel

Searching

Action	*Effect*
Command-Shift-G	Reverses the direction of Find Same command
Command-Shift-H	Reverses the direction of Find Selection command
Command-Shift-T	Reverses the direction of Replace Same command
Double click	Selects a word
Triple click	Selects a line

In the "Replace" Dialog

Holding down Shift while selecting OK reverses the direction of Find and find-and-replace operations.

Other Tips

Double-clicking on any of the characters (,),[,],{,},',",/,\,` selects everything between the character and its mate.

Holding down Option while selecting "Tile Windows" or "Stack Windows" includes the Worksheet in the tiling or stacking operation.

Holding down Option while pressing Return disables auto-indent for that line.

Holding down Option while pressing Enter invokes the Commando on a command line. (For information about the Commando command, see Chapter 3.)

▶ The Help Hotline

One tip deserves its own heading: How to use MPW's Help command. Help is an online help utility that you can summon simply by entering the Help command, either with or without parameters. If you type just the word

```
Help
```

without any parameters, MPW gives you a list of the parameters you can use with the Help command. One of those parameters is the word "Commands." If you use the Commands parameter by executing the command

```
Help Commands
```

MPW lists all the commands that it can help you with. Be prepared for a long list; it will include all of MPW's 120+ commands.

You can obtain summaries of various kinds of help that are available by entering the commands listed in Table 2-10.

Table 2-10. Commands for obtaining help summaries

Command	*Result*
Help [*commandName*]	Information about *commandName*
Help Commands	A list of commands
Help Expressions	A summary of expressions
Help Patterns	A summary of patterns (regular expressions)
Help Selections	A summary of selections
Help Characters	A summary of MPW shell special characters
Help Shortcuts	A summary of MPW shell shortcuts
Help Variables	A summary of the standard MPW shell variables
Help Projector	A summary of Projector, a project management system

The Evaluate Command

Listing 2-1 illustrates the use of the Help command. It shows what MPW writes to the screen when you execute the command

```
Help Evaluate
```

Listing 2-1. Listing returned by the Help Evaluate command

```
Evaluate             # evaluate an expression
Evaluate [-h | -o | -b] [word...]   > value
Evaluate Name [binary operator]= expression
    -h  # display result in hexadecimal (leading 0x)
    -o  # display result in octal (leading 0)
    -b  # display result in binary (leading 0b)
```

The Evaluate command is an important command that MPW uses both to evaluate expressions and to perform mathematical operations. As shown in Listing 2-1, its syntax is

Evaluate [-h | -o | -b] [*word*...]

where the parameter *word* equates to the expression being evaluated.

When Evaluate is used in an MPW script, it is often delimited by the backquote character (`). As mentioned earlier in this chapter, the backquote is MPW's command substitution character. When a command enclosed in backquotes is called by another command, it passes its output back to the command that called it. The calling command then uses the output of the backquoted command as a parameter.

For example, when you execute the command

```
Evaluate 1 + 1
```

Evaluate echoes its output

```
2
```

to standard output, normally the screen.

The preceding example would be more useful if the parameters of the Evaluate command were variables rather than constants. Evaluate can perform operations on variables, as in this example:

```
Set a 2
Set b 5
Evaluate {a} + {b}
```

The output of this set of command lines is, of course, 7.

Setting Variables with the Evaluate Command

You can also use the output of the Evaluate command to set the value of a variable. You can then use the variable in other Evaluate commands. For example, the command

```
Set x `Evaluate {a} + {b}`
```

sets the variable {x} to the sum of {a} + {b}.

Evaluating String Expressions

Evaluate can be used to evaluate string expressions, as well as to perform mathematical operations. For example, if you execute these commands

```
Set alpha a
Set beta b
Evaluate "{alpha}" =~ /{beta}/
```

the Evaluate command echoes the output

```
0
```

which is MPW's value for "false," because the strings "alpha" and "beta" are not the same.

Evaluate also works with variables when it is used to evaluate strings. For example, the output of this pair of commands

```
Set w "alpha"
Evaluate "{w}" =~ /"alpha"/
```

is 1, or "true," because the evaluated strings match.

Several features of this last example are worth pointing out. First, note that in the second line, the {w} variable is enclosed in quotation marks. This means that if the string equating to the variable contained blanks, the command would still work.

Next, notice that the =~ operator is used to compare the two strings. In MPW, the =~ and !~ operators are used to test whether two strings match or not, whereas the == and != operators are used for the same purposes in arithmetical comparison operations, and in case-sensitive string comparisons.

Finally, note that slash bars—not quotation marks—are used to delimit the Evaluate command's second parameter. That is because slash bars are regular expression delimiters, and the Evaluate command must search forward—on its own command line—for a match. This is an unusual use for the delimiters /.../, which are seen more often enclosing the parameters of the Find, Replace, and Search commands.

More examples illustrating the use of the Evaluate command are presented later in this chapter.

▶ Writing MPW Commands

▶ Typing and Entering Commands

Now that you have seen how the command language works, you may be interested in executing some MPW commands. One way to start is to use a set of commands to clear the text from a window. In this exercise, you will execute four commands: Duplicate, Open, Clear, and Close.

▶ Clearing a Window

When you launch MPW for the first time, the first thing displayed on the screen is a Worksheet window—a file that contains several pages of information about MPW. The text was apparently placed there by someone who wanted to create a "README" document for first-time MPW users and wanted to make sure that they noticed it. That may have been a good idea; when you start using a program as complex as MPW, you probably need all the help you can get. And the MPW startup window does contain some information that you might want to look at from time to time. But once you know your way around in MPW, chances are that you will want to start off with a clean Worksheet each time you launch MPW, not a window full of information about the MPW system.

Fortunately, you can easily clear the README text from your startup screen without losing it forever. All you have to do is copy it into some other file where it is safe, and then erase it from your Worksheet window using the Duplicate, Open, Clear, and Close commands. And, as it happens, that's an operation that you can perform right now, with the help of MPW's command-line interpreter.

1. Using the horizontal scroll bar on the right-hand side of your Worksheet window, scroll all the way down to the bottom of the startup text. Then, just below the last line of the text, type this line:

   ```
   Duplicate Worksheet Worksheet.Orig
   ```

   Next, press the Enter key on your numeric keypad (not the Return key on the main part of your keyboard). MPW then copies the startup text that is on your Worksheet into a new text file named Worksheet.Orig.

2. To verify that MPW has indeed carried out the command you issued, type the command line

   ```
   Open Worksheet.Orig
   ```

   and then press Enter.

   MPW responds to this command by opening a document window titled Worksheet.Orig, which should contain the text that you have copied into your newly created Worksheet.Orig file.

3. Now that you know that the startup text in your Worksheet window has been copied into a new text file called Worksheet.Orig, you can clear the text from your Worksheet window screen. At the bottom of your Worksheet.Orig window, use the • (Option-8) and ∞ (Option-5) symbols to type the command

   ```
   Clear •:∞
   ```

   and press the Enter key. The startup text in your Worksheet window then disappears. Bring your Worksheet window to the front to see if this exercise worked. If it did, you now have an empty Worksheet window, and the text that used to be in it is now safely stored away in the Worksheet.Orig file.

   Let's examine how the exercise worked. When the special • character (Option-8) is used alone in an MPW command, it represents the beginning of a text file. The ∞ symbol (Option-5) represents the end of a text file, and the colon (:) used between the two characters represents everything in between. So the command

```
Clear •:∞
```

means "Clear everything (from the document in the target window)."

Note that the • and ∞ characters—like many special characters in the MPW command language—mean different things when they are used in different contexts. For example, when the • character is used inside slash delimiters, it stands for the beginning of a line, rather than the beginning of a file. Similarly, when the ∞ character is used inside slash delimiters, it denotes the end of a line, rather than the end of a file. Thus the command

```
Find /•charlie/
```

searches for a line that begins with the string "charlie," and then selects, or highlights, any such string it finds. The command

```
Find /charlie∞/
```

finds and highlights the string "charlie" if the string appears at the end of a line.

When the ∞ character follows an option that calls for a number of lines, it means "an unlimited number." Hence, the command

```
Replace -c ∞ /charlie/ "zebra"
```

replaces the string "charlie" with the string "zebra" every time it occurs in a file.

4. When you have copied your Worksheet window to the Worksheet.Orig window, you can remove the Worksheet.Orig window from your screen by either clicking in its close box or typing

   ```
   Close Worksheet.Orig
   ```

   in your Worksheet window.

MPW responds to the Close command by displaying a dialog box that asks you if you want to save the text in the Worksheet.Orig window. The answer is yes, so click the "Yes" button. MPW then removes the Worksheet.Orig window from your screen.

Important ▶

The Replace Command. The Replace command, as its name indicates, replaces strings or expression in a file with other strings or expressions. Its syntax is

```
Replace [-c count] selection replacement [window]
```

where the *count* option is a number specifying how many occurrences of the *selection* parameter are to be replaced. Thus to replace 100 occurrences of the string "charlie" with the string "zebra," the command would be

```
Replace -c 100 /charlie/ "zebra"
```

If no -c option is specified, only the first occurrence of the word "charlie" is replaced.

▶ Options and Parameters in MPW Commands

In the MPW command language, commands can be issued with or without options and parameters. The shortest kind of MPW command consists of a single word. For example, the command

```
Beep
```

causes the loudspeaker on your Macintosh to emit a sound. The command

```
Date
```

prints the date and time on your screen, in this format:

```
Friday, April 6, 1991 1:24:45 PM
```

You can also place an option—a letter preceded by a negative sign—after the Date command. For example, if you type

```
Date -d
```

MPW displays the date, without the time, like this:

```
Friday, April 6, 1991
```

Other options that can be used with the Date command are listed in Appendix A.

Using Parameters with the Beep Command

You cannot use options with the Beep command, but you can use parameters. By using parameters, in fact, you can make Beep do much more than generate a simple sound. The syntax of the Beep command is:

```
Beep [note [,duration [,level]]]…
```

Beep's *note* parameter is either a number indicating the count field for the square wave generator—a mechanism described in the chapter titled "Summary of the Sound Driver" in *Inside Macintosh*—or a string in this format:

```
[ n ] letter [ # | b ]
```

The *n* variable in this string is an optional number between –3 and 3 that specifies a number of octaves below or above middle C. The *letter* option specifies the note (A–G) to be played and is followed by an optional sharp (#) or flat (b) sign. Any sharps (#) that are used must be enclosed in quotation marks. Otherwise, MPW interprets them as comment characters.

The *duration* parameter, which is optional, is given in sixtieths of a second, with the default being 15. The *level* is a number from 0 through 255, with a default of 128.

For each parameter given, Beep produces the specified note for the specified duration and sound level. As you have seen, if no parameters are passed, a simple beep is produced. But, if you enter this two-line command

```
Beep 2E,40 '2C,40' 2D,40 1G,80
Beep 1G,40 2D,40 2E,40 2C,80
```

Beep plays a familiar melody.

▶ Writing a Script

To turn one or more command lines into a script, or a command file, all you have to do is save what you have written into a file, and then execute the file by typing its name as a command. As an example, let's take the two command lines in the preceding example and save them as a command file.

First, type the two lines into your MPW Worksheet window, like this:

```
Beep 2E,40 '2C,40' 2D,40 1G,80
Beep 1G,40 2D,40 2E,40 2C,80
```

Now select both lines; that is, highlight them by clicking and dragging your mouse. Copy the lines you have selected onto the Clipboard by typing Command-C. Then open a new window by typing Command-N. When you're prompted for a window name, name your new window Chimes.

When your new window appears, paste your commands into it by typing Command-V. When you see your command lines in the window, save it by typing Command-S and close it by typing Command-W.

Now type the Chimes command into your worksheet window. If you hear the chimes, congratulations! You have just written a script—and created a command!

By the Way ▶

Sounding Off. With the Beep command, you can write scripts that play theme songs when MPW performs specified operations. For example, in the modified UserStartup script presented in Chapter 3, MPW imitates Big Ben—and then prints a time stamp on the screen—to signal that MPW has finished loading and an editing session has begun.

▶ Redirecting Input and Output

By default, most MPW commands read their input from standard input, write their output to standard output, and write any errors that they may encounter to diagnostic output. MPW's standard input is text typed in a window, and its standard and diagnostic output appear following commands that are typed on the screen. MPW's redirection operators are shown in Table 2-11.

Table 2-11. MPW Redirection Operators

Operator	*Meaning*
< *name*	Standard input is taken from *name*.
> *name*	Standard output replaces the contents of *name*. The *name* file is created if it doesn't exist.
>> *name*	Standard output is added to the contents of *name*.
≥ *name*	Diagnostic output replaces the contents of *name*. The *name* file is created if it doesn't exist.
≥≥ *name*	Diagnostic output is appended to *name*. The *name* file is created if it doesn't exist.
Σ *name*	Standard output and diagnostic output replace the contents of *name*. The *name* file is created if it doesn't exist.
ΣΣ *name*	Standard output and diagnostic output are appended to *name*. The *name* file is created if it doesn't exist.

MPW defines its standard input and output assignments with a reserved pathname that is used internally. MPW's reserved pathnames are used to identify pseudo-devices, or devices that do not actually exist, instead of to identify actual volume names. The names of pseudo-devices have special meanings when they are used in files opened by the shell (such as files assigning I/O redirection) or in files opened by MPW tools. Names of pseudo-devices are most often used in shell command files.

One pseudo-device, identified as Dev:, is used to identify standard input and output. The pseudopathnames in which Dev: is used are listed in Table 2-12.

Table 2-12. The pseudo-device Dev:

Pseudopathnames	*Meaning*
Dev:Console	Window from which command was executed
Dev:StdIn	Current assignment for standard input
Dev:StdOut	Current assignment for standard output
Dev:StdErr	Current assignment for diagnostic input
Dev:Null	Empty input stream; first-in, never-output stream; the "bit bucket"

You can see what MPW's diagnostic output looks like by entering a command that contains an error. For example, if you type

```
Echo "How are you this morning?    # This won't work
```

the shell responds with the error message

```
### MPW Shell - "s must occur in pairs.
```

You can override MPW's I/O defaults—the keyboard and the screen—by using I/O redirection, that is, by using the <, >, ≤ (Option-<), and ∑ (Option-W) characters. The symbol > causes a command to write to the parameter that follows it, and the symbol < causes a command to read input from the parameter that follows it. For example, if your MPW directory contained a file named Puppy1 and another file named Puppy2, the command

```
Catenate Puppy1 Puppy2 > TwoPuppies
```

would concatenate files Puppy1 and Puppy2 into a combined file named TwoPuppies. (More information about the Catenate command is presented later in this chapter.)

If you had a file called Errors in your MPW directory, and the file contained just one string, "This is an error," the command

```
Alert < Errors
```

would display an alert dialog containing the line, "This is an error"—the contents of the Errors file. More information about the Alert command is presented in Chapter 3.

The ≥ symbol causes diagnostic output to replace the contents of the file that comes after it, whereas the ∑ symbol causes both standard output and diagnostic output to be written to the target file. When the ∑ symbol is used alone to write to a file, the diagnostic information that it writes replaces the previous contents of the file. A pair of ∑ symbols written together (∑∑) causes diagnostic output to be appended to the end of the file. For example, the command

```
Asm -a MyFile.a ∑∑ LogFile
```

assembles MyFile and writes both the output of the assembly and its diagnostic output to the LogFile file.

One use for redirection is to write the desired information to a file, while throwing unneeded output away; that is, to write standard output to a file but discard diagnostic output, or to write diagnostic output but discard standard output. For example, the command

```
Echo "Good Morning" > Dev:Null
```

writes the diagnostic output of the Echo command to standard diagnostic output, but tosses its standard output into the bit bucket.

▶ Variables in MPW Commands

The MPW shell has a number of predefined variables that can be used in commands. It also allows you to declare any number of additional variables by using MPW commands. In MPW, variables are used for

- creating shorthand notations for long names
- providing various kinds of status information
- placing local variables in scripts
- naming parameters used by scripts and tools
- providing shorter names for certain defaults used by the MPW shell

You can define variables with the MPW Set command, and you can remove variable definitions with the Unset command. Once you have defined a variable with Set, you can use it in any command by enclosing its name in curly brackets. For example, the command

```
Set SuperHero Darryl
```

defines a user variable called SuperHero that has the value Darryl. When you have defined a variable in this way, you can obtain its value by using the Echo command, which writes text to standard output, normally the screen. For example, if you enter the command

```
Echo {SuperHero}
```

MPW writes the string

```
Darryl
```

to your screen.

▶ Kinds of MPW Variables

MPW has a number of shell variables, or variables that are defined every time the shell is launched. MPW uses shell variables to define the pathnames of important files and directories and to set up defaults such as the typeface and type size that are used to display text in windows.

Some shell variables are called predefined variables. They equate to pathnames used by MPW, and they are defined and maintained by the shell. Examples of predefined variables are {Boot}, which always equates to the volume name of your boot disk, and {Target}, which is always the full pathname of the Worksheet window.

Other shell variables are known as startup variables. They include {MPW}, which is defined as the volume or folder that contains MPW, and {Libraries}, which equates the name of the directory that contains libraries shared by MPW's compilers. Startup variables are defined in the MPW UserStartup script, and you can redefine them to suit your own needs and preferences.

There are also several other kinds of MPW variables. The kinds of variables used in MPW were listed in Table 2-5.

▶ Startup Variables

Startup variables are defined in a command script called the Startup script, which is executed every time MPW is launched. By default, the Startup script sets {MPW} to the directory that contains the MPW shell. If you move MPW out of its folder and onto your desktop, you must change the value of {MPW} in the MPW Startup file. Procedures for modifying the Startup file are described later in this chapter. You can obtain the current values of shell variables by using the Echo command. For example, if you execute the command

```
Echo {MPW}
```

and your MPW application is stored in a directory called MPW 3.2 on a hard disk named HD, MPW responds by writing the line

```
HD:MPW 3.2:
```

to your screen.

Similarly, if you enter the command

```
Echo {Boot}
```

and the name of your Boot volume is HD, MPW answers

```
HD:
```

Table 2-13 lists the most important variables used by MPW.

Table 2-13. Variables used by MPW

Variables Set in the Startup Script

Variable	*Value*
{MPW}	Full pathname of the Macintosh Programmer's Workshop
{Commands}	List of directories to search for commands
{Echo}	Control the echoing of commands to diagnostic output
{Exit}	Control script termination based on {Status}
{Test}	Control execution of tools and applications
{AutoIndent}	Auto indent setting used for new windows
{CaseSensitive}	Control case sensitivity for searching
{Font}	Font used for new windows
{FontSize}	Font size used for new windows
{Tab}	Tab size used for new windows
{SearchBackward}	Control direction of searching
{SearchType}	Control type of searching (literal/word/expression)
{SearchWrap}	Control wrap-around search
{WordSet}	Set of characters that constitute a word
{AIncludes}	Directories to search for assembly language include files
{CIncludes}	Directories to search for C include files

Table 2-13. Variables used by MPW (continued)

Variables Set in the Startup Script (continued)

Variable	*Value*
{CLibraries}	Directory containing C library files
{Libraries}	Directory containing shared library files
{PInterfaces}	Directory containing Pascal interface files
{PLibraries}	Directory containing Pascal library files
{RIncludes}	Directory containing Rez include files
{Commando}	Name of the Commando tool

Predefined Variables

Variable	*Value*
{Active}	Full pathname of current active window
{Aliases}	List of all defined aliases
{Boot}	Volume name of the boot disk
{Command}	Full pathname of the last command executed
{ShellDirectory}	Full pathname of the directory that contains the MPW Shell
{Status}	Result of the last command executed (0 means successful)
{SystemFolder}	Full pathname of the system folder
{Target}	Full pathname of the target window
{User}	Current user name (initialized to the "Chooser" name)
{Windows}	List of current windows
{Worksheet}	Full pathname of the Worksheet window
{DirectoryPath}	List of common directories to speed changing directories

Table 2-13. Variables used by MPW (continued)

User Variables

Variable	*Value*
{NewWindowRect}	Window rectangle used for new windows (top, left, bottom, right)
{StackOptions}	Options used by the Stack Windows menu command
{TileOptions}	Options used by the Tile Windows menu command
{ZoomWindowRect}	Window rectangle used for a zoomed window (top, left, bottom, right)
{IgnoreCmdPeriod}	Control use of Command-. during critical sections; default value is 1 (Command-. ignored during critical sections of MPW operations)

Parameter Variables

Variable	*Value*
{0}	Name of the currently executing script
{1}, {2}, ..., {*n*}	First, second, and *n*th parameter to the script
{#}	Number of parameters
{Parameters}	Equivalent to {1}, {2}, ..., {*n*}
{"Parameters"}	Equivalent to "{1}," "{2}," ..., "{*n*}"

▶ Defining Variables with the Set Command

This is the syntax of the Set command, which is used to define variables:

```
Set variable value
```

In a Set command, the *variable* named in the first parameter is set to the *value* specified in the second parameter. For example, the command

```
Set MyName Patrick
```

defines a variable called MyName, and sets its value to Patrick.

Once you have executed this command, you can use the {MyName} variable in place of the word "Patrick" in any MPW command.

When you want to substitute a variable for a value, however, you must remember to enclose the name of the variable in curly brackets. For example, if you have executed the command

```
Set MyName Patrick
```

and then you execute the command

```
Echo {MyName}
```

MPW writes

```
Patrick
```

to the screen.

But, if you enter the command

```
Echo MyName
```

MPW responds with

```
MyName
```

rather than

```
Patrick
```

because you have not enclosed the name of the variable in curly brackets.

You can also use the Set command with just one parameter—the name of a variable that you want identified. MPW then writes the value of the variable you have specified to standard output. For example, if you execute the command

```
Set MyName
```

MPW displays the line

```
Set MyName Patrick
```

The Unset Command

You can delete variable definitions with the Unset command. Its syntax is:

```
Unset [name...]
```

where *name* is the variable to be deleted.

Warning ▶

Be Careful with the Unset Command. When you use Unset, you must be careful to specify the name of a variable as a parameter. If you use the Unset command without any parameters, MPW deletes *all of its variable definitions*—and, unless you are sure that this is what you want, it could be a disaster.

Parameter Variables

MPW has a special set of parameter variables, or variables that equate to parameters in scripts. MPW's parameter variables are listed in Table 2-13, presented earlier in this chapter.

One parameter variable, {0}, is always set to the name of the script currently being executed. Numbered variables that have digits other than zero between the curly brackets—listed in the table as {1}, {2}, ..., {*n*}—equate to the first, second, and *n*th parameter to the currently executing script.

The {#} variable equates to the number of parameters in the script currently being run.

The {"Parameters"} variable is equivalent to all of the numbered parameters, with each one enclosed in quotation marks, that is, "{1}" "{2}" ... "{*n*}". It can be used to retrieve all of the parameters in a script when the exact number of parameters is not known, and when some of the parameters may contain spaces and thus should be enclosed in quotation marks.

Using Parameter Variables

To observe how parameter commands work, type this command in a new window and save it into a file named Hex:

```
Echo `Evaluate -h {1} + {2}` >> "{Active}"
```

Then type this line into your Worksheet window, and execute it:

```
Hex 0x20 0x20
```

If MPW prints the result of this calculation on your screen, you have just written a program that creates a hexadecimal calculator! And it isn't a bad one, either. You can add two numbers easily, without having to abide by the contorted syntax of MPW's Evaluate command. You can enter numbers in hexadecimal notation (preceded by either the prefix 0x or the prefix $); in binary notation (using the prefix 0b); in octal notation (with the prefix 0); or in decimal notation (with no prefix at all). Your hex calculator always prints its results in hex. Most important, you now know how to use parameter variables—and you have written a command that accepts variables and uses them.

Improving Your Hexadecimal Calculator

The Hex script could be improved in many ways. As it stands now, all it can do is add. But if you expand the script to read as shown in Listing 2-2, you can use it to perform a number of different kinds of arithmetical and logical operations.

Listing 2-2. A hexadecimal calculator

```
If `Evaluate "{2}" =~ /Plus/`
    Echo `Evaluate -h {1} + {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /Minus/`
    Echo `Evaluate -h {1} - {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /Times/`
    Echo `Evaluate -h {1} * {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /DivBy/`
    Echo `Evaluate -h {1} ÷ {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /ANDWITH/`
    Echo `Evaluate -h {1} AND {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /OR/`
    Echo `Evaluate -h {1} || {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /SHIFTL/`
    Echo `Evaluate -h {1} << {3}` >> "{Active}"
Else if `Evaluate "{2}" =~ /SHIFTR/`
    Echo `Evaluate -h {1} >> {3}` >> "{Active}"
End
```

For example, the command

```
Hex 0x1 OR 0x2
```

now yields the output

```
0x3
```

which is the result of performing a logical OR operation on the numbers 0x1 and 0x2.

You may notice that in the Hex script, the words "AndWith" and "DivBy" are used as variables standing for operators instead of the more conventional words "AND" and "DIV." That's because AND and DIV are reserved words that are used as real operators by MPW.

You also may notice that the Hex script does not run very fast; MPW is not a superpowered command evaluator. But Hex does provide a simple illustration of how parameter variables and structured constructs are used in the MPW command language.

More information about MPW's string, arithmetical, and logical operators is presented in Chapter 4.

▶ Scopes of Variables

MPW can recognize variables that were initialized or modified by the Set command only while they are in the current context. If you use the Set command interactively—that is, in a command line while typing at the keyboard—then the variable that you have set is recognized only by other commands you enter at the keyboard. Similarly, if you use Set to initialize a variable in a command file, or script, MPW recognizes the variable only in that script.

When you initialize variables by typing them in on a command line, they also have a short lifespan; they are wiped out of memory as soon as you exit MPW. However, one set of variables is initialized every time you launch MPW, and you can customize it to suit your own needs and preferences. The procedures for finding and changing these variables are explained later in this chapter.

▶ Extending the Scope of a Variable

You can extend the scope of a variable by using the Export and Execute commands.

The Export Command

The syntax of the Export command is

```
Export name...
```

where *name* is a variable or list of variables to be exported.

When you use Export in a script, the parameter *name* is exported to all scripts enclosed by the script in which you execute the Export command. In other words, variables exported by the Export command are visible to nested scripts.

The Export command does not work in reverse, so you cannot use it—or any other command—to extend the scope of a variable to an enclosing script.

If you want to define a variable globally so that it is recognized by all scripts and by all commands typed interactively on command lines, you can define it in the MPW Startup script or your own UserStartup script, and then export it to nested scripts using the Export command. The Startup script and the UserStartup script enclose all other scripts, so your variable is then global; it is visible to all other scripts and to the MPW command interpreter, which processes command lines.

For example, you could include these lines in your UserStartup script:

```
Set AIncludes "{MPW}Interfaces:AIncludes:"
Export AIncludes
```

In this pair of commands, a variable called {AIncludes} is created and defined as the pathname {MPW}Interfaces:AIncludes, and it is made available to all scripts and programs running under MPW.

The Unexport Command

The Unexport command reverses the effects of Export; if you have used the Export command to export a variable or a list of variables, you can use the Unexport command to remove any desired variables from the exported list. Unexport has the same syntax as the Export command.

The Execute Command

The syntax for the Execute command is

```
Execute script
```

where *script* is the name of the script to be executed.

In MPW, you can run a script simply either by inserting its name into another script or by entering its name as a command on a command line. However, if you merely type the name of a script to run it, the script's variable definitions, exports, and aliases are local in scope and no longer exist after the script has been executed.

If you want the variables in a script to be recognized by nested scripts and by the MPW command interpreter, you should run the script in which the variables are defined using the Execute command, rather than by simply typing the script's name. If the script that you execute in this way exports its variables with the Export command, they then become visible to the script that contains the Execute command.

When you run a script using the Execute command, the script is executed as if its contents appeared on your command line instead of the Execute command. That means that the file executes in the current scope, rather than in its own scope. Another way of expressing this idea is to say that the context in which MPW is operating changes to the context of the executed file. The variables and aliases defined and exported in the executed script thus become visible to the script that contains the Execute command. In addition, the executed script's variable definitions and aliases continue to exist after it finishes executing, rather than disappearing after it has been run.

When you have made changes in a Startup or UserStartup script, you should always run it using Execute to test the changes you have made. If you run an altered Startup or UserStartup script by simply typing its name, any new variables or aliases that you have added to your command script cease to exist after they have been executed.

For this reason, the command that the Startup script uses to call the UserStartup script is

```
Execute "{ShellDirectory}UserStartup"
```

rather than simply

```
"{ShellDirectory}UserStartup"     # this isn't safe
```

If the latter form were used, the Set, Export, and Alias commands defined in the UserStartup script would have no effect.

▶ More About the Echo Command

The Echo command has been used in many examples in this chapter, and it is such an important command that it could have been used in many more. In MPW, Echo is the command that is most often used to pass literal strings to commands. So let's now take a closer look at the Echo command.

Echo takes one or more parameters that are separated by spaces and may optionally be enclosed in quotation marks. The command writes its parameters, followed by a Return, to MPW's standard output (the active window, if no I/O redirection is used). If you do not use any parameters, Echo writes only a Return.

The syntax of the Echo command is:

```
Echo [-n] [parameter...] [> parameters...]
```

When the -n option is used with the Echo command, it means, "Don't write a Return following the parameters." That means that the insertion point—the location of the text cursor—remains at the end of the output line after the text to be echoed is written. The -n option itself is not echoed.

To see how Echo works with a one-word parameter, just enter the command

```
Echo hello
```

Echo then writes

```
hello
```

to your screen.

If you want to pass a string to Echo, you can enclose it in quotes. For example, if you enter

```
Echo "Hello, world"
```

Echo writes

```
Hello, world
```

on your screen.

As you have seen, Echo can also be used with variables. For example, if you type the command

```
Echo {Status}
```

Echo writes the current value of the {Status} variable, that is, the status of the last command executed.

Another way to use Echo is to issue a command like this:

```
Echo ≈.a
```

In response to this command, Echo writes the names of all files in the current directory that end with ".a". Issuing this kind of command might be a good precaution to take before executing a potentially dangerous command—such as one to delete files—with the argument "≈.a".

In this example, the ≈ character (Option-X) is a wildcard character—a character that can be substituted for other characters in a command. Specifically, the ≈ character stands for any string of zero or more characters. When it is used in a file name, as in this example, it is known as a file name generation operator. File name generation operators are described earlier in this chapter and were listed in Table 2-7. More information about wildcard characters is presented later in this chapter, and in Chapter 4.

Echo can be used with redirection operators. For example, enter this command:

```
Echo -n > EmptyFile
```

If *EmptyFile* exists, the Echo command deletes its contents; if the file does not exist, it is created.

▶ The Quote Command

The Quote command, like the Echo command, writes its parameters, separated by spaces and terminated by a Return, to standard output. When Quote is used, however, parameters containing characters that have special meaning to the shell's command interpreter are placed inside single quotation marks. If no parameters are specified, only a Return is written.

Quote is identical to Echo, except for its treatment of parameters that contain special characters. Quote is especially useful when you want to use shell commands to write a script.

These are the special characters that the Quote command places inside quotation marks:

```
Space, Tab, Return and Null, plus:
# ; & | ( ) ∂ ' " / \ { } ` ? ≈ [ ] + * « » ® < > ≥
```

The meanings of these characters are explained in Chapter 4.

Consider the following example of how the Quote command is used. Suppose you had a file on a disk called My Program (note the space between the words). If you entered the command

```
Echo ≈.a
```

MPW would print the names of all the files on the disk that ended in ".a". For example,

```
Sample.a Count.a My Program.a      # File name not in
                                   # quotes
```

But if you entered

```
Quote ≈.a
```

the result would be

```
Sample.a Count.a 'My Program.a'     # File name in
                                    # quotes
```

Note that since the output of the Quote command, unlike the output of Echo, is enclosed in quotation marks, it could be used as a parameter in another command.

By the Way ▶

An Easy Way to Look Up Special Characters. Although there are plenty of tables around that can show you the keyboard equivalents of special characters, there is an easy way to find out how to type a special character without consulting a table. Just go to the Apple menu on your menu bar and select the desk accessory called "Key Caps." It shows you a Macintosh keyboard. Press the Option key, and the keys on the Key Caps keyboard change to the special characters that they print when the Option key is held down. Press Option-Shift, and all the Option-Shift characters are displayed. The Key Caps keyboard, in all three of its modes, is shown in Figures 2-7 through 2-9.

You can also use the Key Caps desk accessory to change fonts and to perform the standard Macintosh cut, copy, and paste commands. When you change fonts with the Key Caps desk accessories, the keys on the DA change from their previous font to the font you have selected. So you can use Key Caps to see what special keyboard characters are available in any font you may want to select.

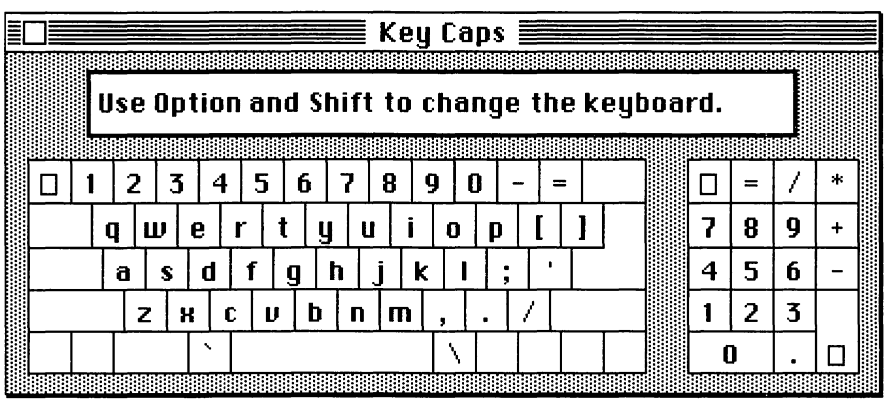


Figure 2-7. The Key Caps DA with no keys pressed.

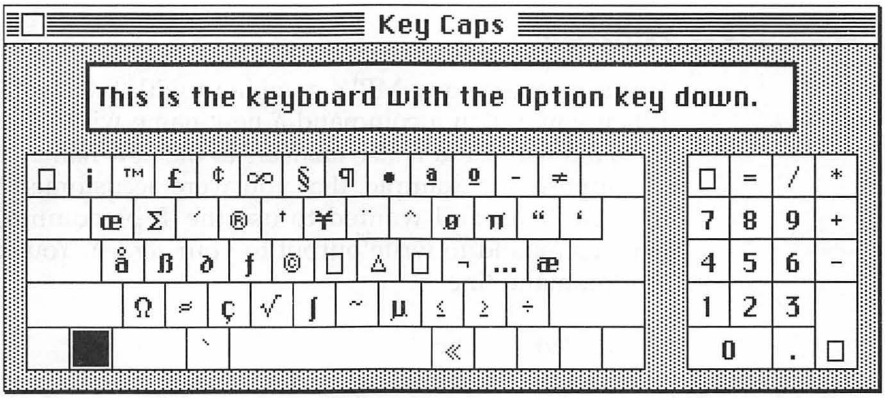


Figure 2-8. The Key Caps DA with the Option key pressed.

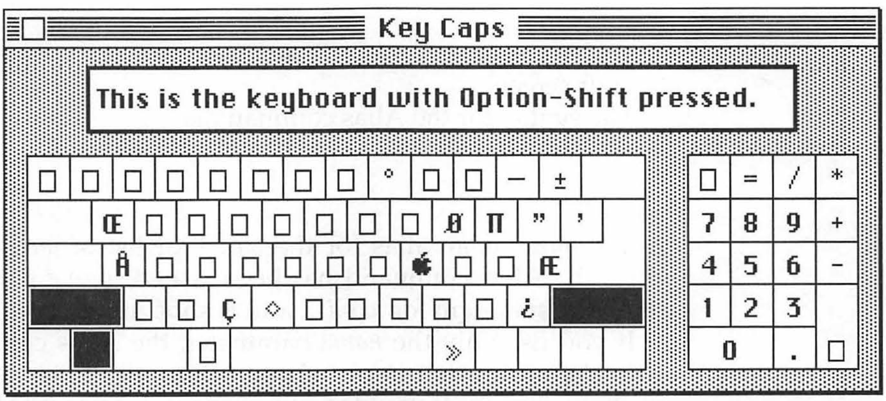


Figure 2-9. The Key Caps DA with the Option and Shift keys pressed.

Aliases

You can rename any MPW command with the shell's Alias command. When you assign a command a new name with Alias, it still responds to its old name, but it also answers to the new name you have given it.

Suppose, for example, that you were accustomed to using MS-DOS or VAX/VMS and wanted to use the Type command instead of the Echo command to write output to your screen. You could simply enter the command line

```
Alias Type Echo
```

and a new command name, Type, would be created. You could then enter the command line

```
Type Hello there
```

and MPW would write

```
Hello there
```

to your screen.

The syntax for the Alias command is

```
Alias [name [word...]]
```

where *name* is an alias for the *word*, or list of *words*, that follows it. Once the Alias command has been issued, *name* is recognized by the shell as a synonym for *word*... and is substituted in its place.

If you use only the *name* parameter, the Alias command writes any alias definition that is associated with *name* to standard output, normally the screen. If you do not use any parameters, Alias writes a list of all aliases and their values.

The Unalias Command

Everything that can be done with Alias can be undone with MPW's Unalias command.

When you want to delete an alias, all you have to do is enter the command

```
Unalias name
```

and the alias specified in the parameter *name* is deleted.

It can be a dangerous practice to use an Alias command without any parameters. If you enter the command

```
Unalias
```

without a parameter, the shell deletes *all aliases* currently in effect.

Some additional uses of the Alias and Unalias commands are listed in Table 2-14.

Table 2-14. The Alias and Unalias commands

Command	*Effect*
Alias *name word...*	*Name* becomes an alias for the word or list of words.
Alias *name*	Displays any alias that may be associated with *name.*
Alias	Displays all alias definitions.
Unalias *name*	Removes any alias definition that may be associated with *name.*
Unalias	Removes all alias definitions.

▶ Making an Alias Permanent

If you invoke the Alias command by typing it on a command line, the alias you have created is recognized only by other commands typed interactively and is wiped out of memory altogether as soon as you exit MPW. If that were the only way you could use Alias, it would not be of much value.

Fortunately, there is a way to create aliases that are recognized by MPW in the same way that it recognizes its own native list of command names. You can create an alias, make an alias global, and make it available for your use every time you launch MPW by initializing it in a special script called the **UserStartup** script.

▶ The Startup and UserStartup Scripts

When MPW is launched, it executes a special command script called the Startup script. The Startup script, like all MPW scripts, is a text file that is executed by the shell. The Startup script initializes a set of variables used by MPW, does some other housekeeping work, and then calls a second command script called the UserStartup script.

The UserStartup script builds the Project, Directory, and Build menus. It can also be used to define customized aliases and variables, and customized menus and menu items.

As mentioned earlier, MPW has a number of shell variables that it uses to perform important operations such as searching for pathnames and creating screen displays.

Some shell variables, called predefined variables, contain pathnames that are often used by the shell. Predefined variables are automatically set to their proper values before the Startup script is executed. If their values change while MPW is running, their definitions are changed automatically. There is no way that you can change their values, and there is no reason that you would want to try.

Some of MPW's predefined variables are {Boot}, which always contains the name of the boot disk; {SystemFolder}, which is always set to the directory that contains the System Folder and Finder; and {ShellDirectory}, which is always set to the directory that contains the MPW shell. The predefined variables used by MPW are listed in Table 2-15.

Table 2-15. Predefined variables

Variable	*Definition*
{Boot}	The boot disk
{SystemFolder}	The directory that contains the System Folder and the Finder
{ShellDirectory}	The directory that contains the MPW shell
{Active}	The active (topmost) window
{Target}	The target (second topmost) window
{Worksheet}	The name of the Worksheet window
{Status}	The result of the last command executed (zero if no error, nonzero if an error was returned)
{User}	Automatically defined to the name that appears in the Chooser

Other shell variables, called startup variables, are defined in the Startup script. Some of the startup variables are {MPW}, which equates to the volume or folder that contains MPW; {Commands}, which contains a list of directories to search for commands; and {Libraries}, which equates to a directory that contains libraries shared by MPW's compilers.

Startup variables, unlike predefined variables, can be redefined by the MPW user. You can set any startup variable to any value you like by modifying the UserStartup script. MPW's startup variables are listed in Table 2-16.

Table 2-16. Variables defined in the startup script

Variable	*Definition*
{MPW}	The volume or folder containing the Macintosh Programmer's Workshop. If you move the MPW Shell to the desktop, you should redefine this variable to be "{Boot}MPW:" Initial setting is MPW"{ShellDirectory}".
{Commands}	Directories to search for commands. Initial setting is ":,{MPW}Tools:,{MPW}Scripts:".
{AIncludes}	Directories to search for assembly language include files. Initial setting is "{MPW}Interfaces:AIncludes:".
{Libraries}	Directory that contains shared libraries. Initial setting is "{MPW}Libraries:Libraries:".
{CIncludes}	Directories to search for C include files. Initial setting is "{MPW}Interfaces:CIncludes:".
{CLibraries}	Directory that contains C libraries. Initial setting is "{MPW}Libraries:CLibraries:".
{PInterfaces}	Directories to search for Pascal interface files. Initial setting is "{MPW}Interfaces:PInterfaces:".
{PLibraries}	Directory that contains Pascal libraries. Initial setting is "{MPW}Libraries:PLibraries:".
{RIncludes}	Directory that contains Resource include files. Initial setting is "{MPW}Interfaces:RIncludes:".
{CaseSensitive}	If nonzero, pattern matching is case sensitive. Initial setting is 0.
{SearchBackward}	If nonzero, search goes backwards. Initial setting is 0.

Table 2-16. Variables defined in the startup script (continued)

Variable	*Definition*
{SearchWrap}	If nonzero, search wraps. Initial setting is 0.
{SearchType}	Specifies the default searching type. (0/literal, 1/word, 2/regular expression). Initial setting is 0.
{Tab}	Default tab setting for new windows. Initial setting is 4.
{Font}	Default font for new windows. Initial setting is "Monaco."
{FontSize}	Default font size for new windows. Initial setting is 9.
{AutoIndent}	If nonzero, auto indentation is the default for new windows. Initial setting is 1.
{WordSet}	Character set that defines words for searches and double-clicks. Initial setting is 'a-zA-Z_0-9'.
{PrintOptions}	Options used by the Print Window and Print Selection menus. Initial setting is '-h'.
{Exit}	If nonzero, command files terminate after the first error. Initial setting is 1.
{Echo}	If nonzero, commands are echoed before execution. Initial setting is 0.
{Test}	If nonzero, tools and applications are not executed. Initial setting is 0.

▶ Modifying the Startup Script

One startup variable that you can redefine is {MPW}, which equates to the name of the directory that contains the MPW shell. For example, if you move the MPW shell from its default directory onto your desktop, you must change the value of {MPW} in the Startup file.

You can also make other changes in MPW's directory structure—and if you do, you must modify other startup variables so that MPW can find the files and directories whose locations you have changed.

Note ▶

Should You Modify Your Startup Scripts? Engineers at Apple strongly discourage MPW users from modifying the MPW Startup and UserStartup scripts. As an alternative, Apple says that if you want to modify MPW's startup procedures, you should create an auxiliary UserStartup script (a script with a file name written in the format UserStartup•*fileName*, as explained later in this chapter).

However, many MPW wizards do edit their Startup and UserStartup scripts instead of creating auxiliary scripts. The reason is simple: The more startup scripts you use, the longer it takes MPW to start up when it is launched. So some MPW experts combine their Startup and UserStartup scripts into a single script, as outlined later in this chapter, and don't use any UserStartup script at all.

As you can see by looking at the listings at the end of this chapter and Chapter 3, I have modified both my Startup script and my UserStartup script, and my MPW system works fine. I also have some auxiliary UserStartup scripts, which I use to set up my MPW environment in very special ways from time to time.

Should you modify your Startup and UserStartup scripts? That's up to you. But if you do, be sure to make copies of your original scripts—and store your copies where they will be *very* safe—before you start modifying anything. If you do that, I don't see how you can get yourself into too much trouble.

▶ Modifying the MPW Directory Structure

When you install MPW, the Installer sets up an MPW folder that has exactly the kind of directory structure that the Startup script expects. Consequently, when MPW carries out an operation such as compiling or linking a program, it can find the interface and library files that it needs by using pathname variables that are defined in the Startup script.

Once MPW is installed, you can rearrange the files and folders inside the MPW folder in any way you like. But if you modify the MPW directory structure in this way, you must redefine the startup variables that tell MPW where its directories are. Otherwise, MPW cannot find the files it needs, and it returns an error message when you try to compile a program.

One possible reason for modifying the MPW directory structure might be to categorize the files and folders inside the MPW folder by language. If you were low on disk space (and who isn't?), you could increase the space available on your hard disk by using a language-based directory system.

Suppose, for example, that your MPW system included an assembler, a C compiler, and a Pascal compiler. You could then set up one folder for each language in your system: A folder called AFolder for assembly language interfaces and libraries; a folder called CFolder for C interfaces and libraries; and a folder called PFolder for Pascal interfaces and libraries. You could also add a folder called RFolder to hold Include files used by the Resource Manager (for more information on the Resource Manager, see Chapter 6).

If you arranged your directory structure in this way, you could temporarily remove from your hard disk any language folders that you would not be needing for a while, and you could move them back to your hard disk at any time you needed them again. For example, if you were working on a project written in C and did not plan to do any work in assembly language or Pascal for a while, you could temporarily remove the AFolder and PFolder from your hard disk, which would free about 1-1/2 megabytes of disk space for other uses. Later, if you went back to a project written in Pascal or assembly language, you could restore its folder to your hard disk and perhaps remove your C folder.

Figures 2-10 through 2-12 show how to rearrange the MPW directory structure into a language-based configuration. Figure 2-10 illustrates MPW's original directory structure, and Figure 2-11 shows a directory structure that has been reconfigured into a language-based arrangement. Figure 2-12 shows the contents of the new folders AFolder, CFolder, PFolder, and RFolder, in the language-based configuration.

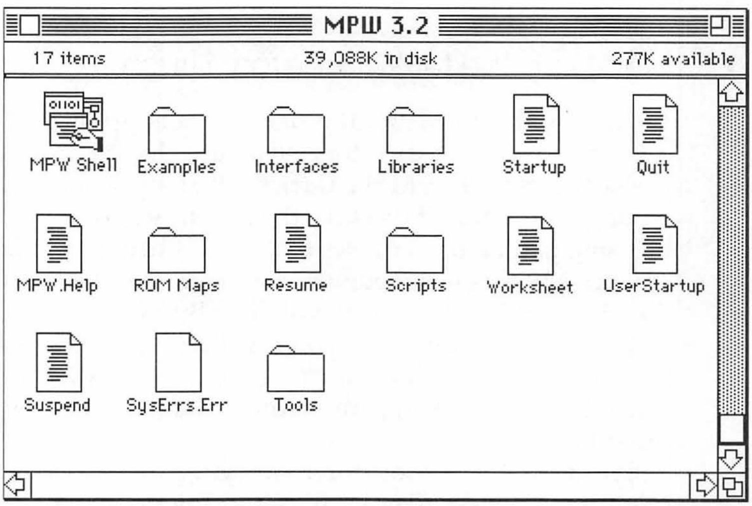


Figure 2-10. Default directory structure

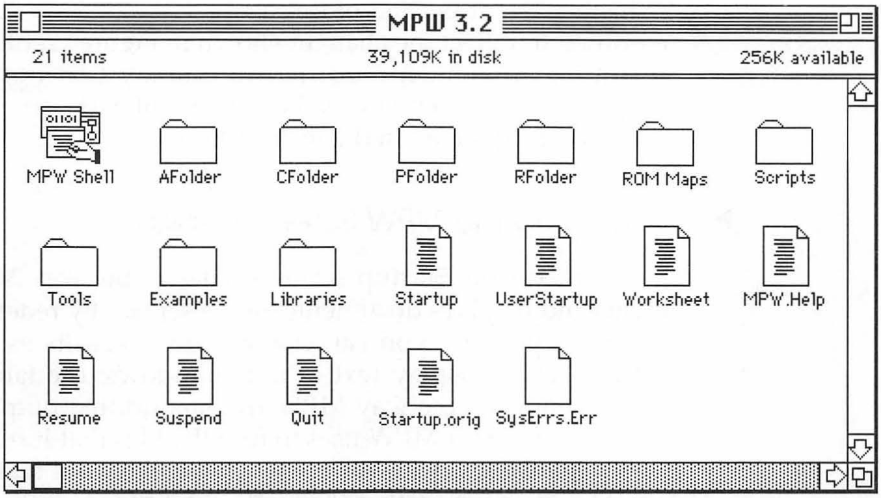


Figure 2-11. Revised directory structure

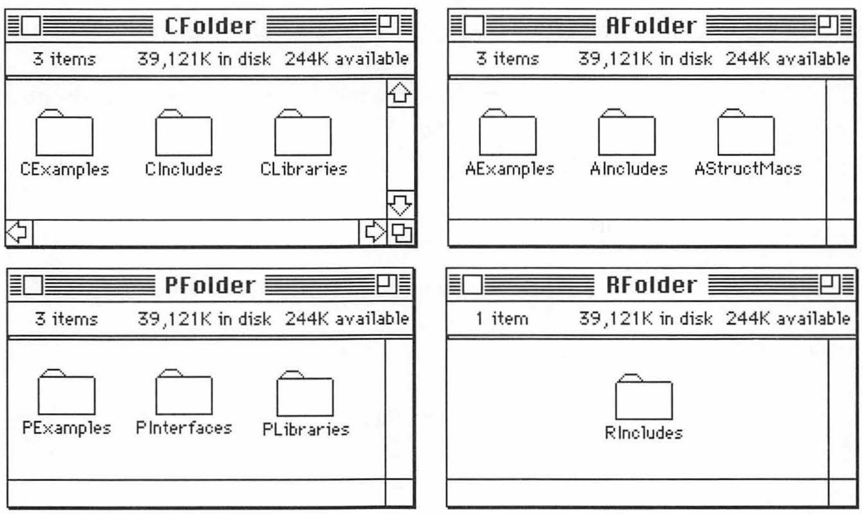


Figure 2-12. Contents of the new directories

Listing 2-4, at the end of this chapter, is a Startup script that has been modified to reflect the changes shown in Figures 2-10 through 2-12. The script has also been modified to display text in new windows in 10–point Courier type instead of in 9-point Monaco. These two modifications are explained in the next section.

▶ Changing the MPW Screen Display

You can edit the Startup script to change the way MPW prints documents and displays documents on the screen. By redefining variables in the Startup script, you can change such defaults as the typeface that MPW uses to display text in new windows; the tab settings used in MPW windows; the way MPW formats printed output; and the directory structure that MPW uses to find the files that it runs.

For example, if you want MPW to display files that you create in 10-point Courier type rather than in 9-point Monaco (the default), you can change the portion of the Startup script that reads

```
#    {Font} - Default Font for new windows.
     Set Font Monaco
     Export Font

#    {FontSize} - Default font size for new windows.
     Set FontSize 9
     Export FontSize
```

to read

```
#    {Font} - Default Font for new windows.
     Set Font Courier
     Export Font

#    {FontSize} - Default font size for new windows.
     Set FontSize 10
     Export FontSize
```

Note ▶

Which Font Should I Use? MPW wizards recommend that you use Monaco, Courier, or some other monospaced font as your default MPW font because the use of fancier but proportionally spaced fonts can make spacing in source code pretty hard to handle.

▶ Redefining the {WordSet} Variable

Another variable that you can modify in your Startup script is {WordSet}, which sets the rules for defining a word. In the MPW Startup script, the {WordSet} variable is defined this way:

```
Set WordSet 'A-Za-z_0-9'
```

According to this definition, a word is composed of any combination of the uppercase letters A through Z, the lowercase letters a through z, the numerals 0 through 9, and the underscore character (_). When {WordSet} is defined in this fashion, double-clicking on a word highlights all the alphabetic and numeric characters that it may contain. Any underscore character in the word is also considered a part of the word. But, if the word contains any special character such as a period or a colon, MPW stops highlighting the word at that point and does not select the special character.

If you want to use double-clicking to select pathnames, which may contain colons and periods, you can redefine the {WordSet} variable in your Startup script with a line like this:

```
Set WordSet 'a-zA-Z_0-9.:'
```

You can then select a complete pathname (as long as it contains no spaces) by double-clicking on it in a window.

▶ Making and Saving Your Modifications

To modify the Startup script, first open it either by selecting Open from the File menu or by entering the command

```
Open Startup
```

You can then use standard Macintosh editing functions to change the values of any of the variables that the Startup script defines. Then you can save the script in its new form.

Before you make any changes in your Startup file, be sure to copy your original Startup script into some other folder for safekeeping, especially if it has been modified.

How Startup Calls UserStartup

Each time the Startup script defines a shell variable, it exports the variable's definition to other scripts by using the Export command, which was described earlier in this chapter.

When all shell variables have been defined and exported, the Startup script executes the UserStartup script with this command:

```
Execute "{ShellDirectory}UserStartup"
```

When the UserStartup script is executed, it builds three menus (Project, Directory, and Build) on the MPW menu bar. Then, if you wish, UserStartup defines any aliases, variables, menus, or menu items that you want to create for your own use. If you modify the UserStartup script by adding definitions of customized aliases, variables, menus, and menu items, they are initialized and ready to use every time you launch MPW.

Modifying the UserStartup Script

You can fine tune MPW to match your own needs and preferences by modifying the UserStartup script. The most common reasons for modifying the UserStartup script are to initialize user-defined aliases and user-defined variables, and to add custom menus and menu items to the MPW menu bar.

The procedures for modifying the UserStartup script are the same as those used to modify the Startup script. Just open your UserStartup script, make your modifications, and save the script in its modified form.

In the next sections, you will get an opportunity to create some aliases and define some variables by modifying the UserStartup script. In Chapter 3, you will see how you can modify the UserStartup script to create your own menus and menu items.

Creating Aliases in the UserStartup Script

MPW has many commands that perform the same functions as commands that are used in other programming environments, but have different names. For example, the MPW Echo command works much like the command that has the name "Type" in UNIX and MS-DOS.

If you have been using a non-Macintosh programming environment, and are accustomed to using the Type command rather than the Echo command, you can teach MPW what Type means by initializing an alias called Type, and defining it as Echo. Then, if you have trouble remembering the MPW Echo command, you can use Type instead.

You can add the alias Type to your UserStartup script by opening the UserStartup script, scrolling down to the bottom of the script, and typing a line like this:

```
Alias Type Echo
```

You can create more aliases by typing in additional alias definitions, one to a line. When you have created all of the aliases you want, you can save your modified UserStartup script, execute it using the Execute command, and start using your new aliases in MPW commands.

Listing 2-3 is a selection of other aliases that you can add to your UserStartup file. They are a potpourri of keyboard shortcuts and UNIX, MS-DOS, and Apple ProDos commands. The complete UserStartup file from which they were taken is presented at the end of Chapter 3. If you are accustomed to using a non-MPW programming environment, you can probably think of a lot of other aliases that you would like to define.

Listing 2-3. Aliases in a modified UserStartup script

```
Alias Type Echo               # write text to standard
                              # output
Alias Dir Files               # list files and directories
Alias CD SetDirectory         # change default (current)
                              # directory
Alias ChDir SetDirectory      # change default (current)
                              # directory
Alias Create New              # open new window (file)
Alias Cpy Duplicate           # copy a file
Alias Dup Duplicate           # copy a file
Alias cp Duplicate            # copy a file
Alias MD NewFolder            # create new directory
Alias MkDir NewFolder         # create new directory
Alias Cls Clear •:∞           # clear screen (target
                              # window)
```

Listing 2-3. Aliases in a modified UserStartup script (continued)

```
Alias ar Lib               # make library file
Alias cat Catenate         # shorter than Catenate
Alias cc 'C -mbg off'      # compile C program, MacsBug
                           # off
Alias cmp Equal            # compare files & directories
Alias diff Compare -b      # compare, ignoring minor...
                           # ...differences in white
                           # spaces
Alias df Volumes -l        # list volumes in long format
Alias expr Evaluate        # evaluate an expression
Alias grep Search          # good old grep
Alias ll Files -x tckrbm   # list files and
                           # directories...
                           # ...in a nice format
Alias lr Files -m 5 -r     # list files, directories...
                           # ...and subdirectories
Alias ls Files -m 5        # list files in 5 columns
Alias man Help             # Help
Alias mv Move              # Move files/directories
Alias pr Print             # Easier to type
Alias rm Delete            # Two letters for six
Alias source Execute       # Execute script in current
                           # scope
Alias tar Backup           # Saves keystrokes
Alias tr Translate         # Saves more keystrokes
Alias wc Count             # Count lines and characters
```

▶ Defining Variables in the UserStartup Script

In the UserStartup script, you can define several user variables that are initialized by MPW at launch time but are not specifically initialized in the Startup script. Although you will not find these variables anywhere in the Startup script, you can still define them yourself. They are

{NewWindowRect}, which sets the coordinates of new windows; {ZoomWindowRect}, which sets the zoom coordinates of new windows; {StackOptions}, which sets parameters for the StackWindows command; {TileOptions}, which sets parameters for the TileWindows command; and {IgnoreCmdPeriod}, which determines whether or not Command-Period, the standard Macintosh "Halt" keystroke sequence, is recognized during critical operations (it's a good idea to stay away from that one).

You can set the values of MPW's user variables and include those values in your UserStartup script by executing a shell script called UserVariables. To run the UserVariables script, you simply execute the command

```
UserVariables
```

When you execute the UserVariables script, it displays a Commando dialog called, logically enough, the UserVariables Commando. A Commando dialog is a special kind of MPW dialog that you can use to execute commands by clicking on controls, instead of typing and entering commands.

With the UserVariables Commando, you can set custom defaults for MPW's user variables and then include those defaults in your UserStartup script. Instructions for using the UserVariables Commando are presented in Chapter 3.

▶ User-Defined Variables

You can also initialize your own user-defined variables in the UserStartup script. A user-defined variable is any variable that you want to make global so you can use it in other scripts. More information about user-defined variables is given later in this chapter.

▶ Creating a Supplementary UserStartup Script

If the thought of modifying your UserStartup script makes you nervous, you can create a supplementary UserStartup script, which MPW runs after your default UserStartup script is executed. You can then use your supplementary script to define customized variables, aliases, and menu items, and you can leave your original UserStartup script unchanged.

MPW has provided you with the option of adding a supplementary UserStartup script by including this set of commands in the Startup script:

```
For __Startup__i in `(Files "{ShellDirectory}" ∂
UserStartup•≈ || Set Status 0) ≥ dev:null`
    Execute "{__Startup__i}"
End
Unset __Startup__i
```

How this block of code works will be explained in Chapter 4. However, you do not have to understand it to use it. To prepare a supplementary UserStartup script, just open a new file in your Worksheet window and give it a name written in accordance with the following example; that is, with a bullet (Option-8) between the word UserStartup and the name you want to give your second UserStartup script:

`UserStartup•`*`scriptName`*

For example, you could call your file

```
UserStartup•MyStartup
```

or

```
UserStartup•Mike
```

Once you have opened a file with that kind of name, save it in the folder where your MPW application resides. The next time you launch MPW, your new file is executed after your original UserStartup script has run. So you can define variables, aliases, menus, and menu items in your supplementary UserStartup file without touching your original UserStartup script.

▶ Running MPW Without a UserStartup Script

If you create a supplementary UserStartup script, you will slightly increase the length of time that it takes to start up MPW. Conversely, MPW starts up faster when you have no UserStartup script at all. And MPW runs perfectly well without a UserStartup script.

If you would like to speed up MPW's loading operation by doing without a UserStartup script, just copy everything that you now have in your UserStartup script onto the end of your Startup script, and you can throw your UserStartup script away. MPW then loads and starts up a little faster because it has only one Startup script to run.

Important ▶

Safe Scripting. You should be very careful, if you decide to combine your Startup and UserStartup scripts into a single file. If you introduce bugs into your Startup and UserStartup scripts when you combine them, you could wind up with problems that are hard to track down. So, before you try to concatenate your Startup and UserStartup scripts, make sure you know what you are doing. And be sure to save copies of both scripts before you begin, especially if you have modified them.

▶ Files and Directories

In dealing with file and directory names, MPW follows the standards of the hierarchical file system (HFS), the file management system currently used on the Macintosh. In this section, the hierarchical file system is dealt with only as it relates to MPW. For more comprehensive information about HFS, refer to the chapter on the File Manager in *Inside Macintosh, Volume IV*.

When you work with MPW, you can manage files and directories without leaving the shell and returning to the Finder or MultiFinder. The easiest way to determine what directory you are in, or to change directories, is to use MPW's Directory menu, as explained in Chapter 3. But you can also perform the operations listed under that menu, and many more, by using command lines. For example, you can set your current directory or find out what the current directory is by using the Directory command. You can obtain lists of directories or of the files in directories by using the Files command. Other commands that are used to list and manage files and directories are shown in Table 2-17.

Table 2-17. File management commands

Command	*Meaning*
Backup	Folder file backup
Catenate	Concatenate files
Close	Close specified windows
Delete	Delete files and directories
Directory	Set or write the default directory
Duplicate	Duplicate files and directories
Exists	Confirm the existence of a file or directory

Table 2-17. File management commands (continued)

Command	*Meaning*
Files	List files and directories
GetFileName	Display a Standard File dialog box
Mount	Mount volumes
Move	Move files and directories
New	Open a new window
Newer	Compare modification dates of files
NewFolder	Create a new folder
Open	Open file(s) in window(s)
Rename	Rename files and directories
Revert	Revert window to previous saved state
Save	Save specified windows
SetDirectory	Set the default directory
SetPrivilege	Set access privileges for directories on file servers
SetVersion	Maintain version and revision number
Target	Make a window the target window
Volumes	List mounted volumes
WhereIs	Find the location of a file
Which	Determine which file the shell executes
Windows	List windows

How MPW Searches for Files

MPW makes no distinction between file names and the names of windows. When you pass MPW a file name as a parameter to a command, it first looks for an open window with the name you have specified. Then, if it does not find one, it looks on a disk for the requested file.

When you pass a command name to MPW, it searches for the command in the directories listed in a shell variable called {Commands}. This search path is initially set to

```
:,{MPW}Tools:,{MPW}Scripts:
```

Since these are the default settings of the {Commands} variable, the shell first assumes that when you type a command, you want to execute a tool. If the shell cannot find such a tool in its {Commands} path, it then assumes that you are looking for a script. Finally, if it cannot find a script, it assumes you want an application.

If you find that this search path slows you down, or you would like to use a different one, you can change it to improve the shell's performance by altering the MPW Startup script as explained earlier in this chapter.

When you use file- or directory-related commands in MPW, you should keep the following rules in mind:

1. The name of a single directory or file cannot be more than 31 characters long.
2. You can use any character except a colon (:) in a file name; you cannot use a colon because colons are used to separate the elements of pathnames.
3. Names of directories and files are not case sensitive, so you can mix uppercase and lowercase letters all you like.

▶ Spaces in File and Directory Names

One important fact to remember about MPW is that it is very sensitive to spaces in file and directory names. If you use spaces in file names without taking special precautions, the MPW shell interprets the words in the file name as individual words in the command language and processes them accordingly. So, when you are dealing with MPW, it is best to avoid using spaces in file and directory names.

If you do use file names with spaces in them, you can avoid confusing MPW by enclosing your file names in quotation marks, like this:

```
Open "HD:Inside MPW:Chapter 1"
```

In the preceding example, the pathname given is a full pathname; that is, it provides the complete pathname of the specified file, all the way back to its root directory. Since a full pathname starts with the name of a disk or volume, it never begins with a colon.

A partial pathname is a pathname that begins its path at the current default directory. Any name that contains no colons or begins with a colon is considered a partial pathname. For example, the name

```
:CExamples
```

is a partial pathname. However, the name

```
HD:
```

is a full pathname, that is, the name of a volume only. You can tell that it is a full pathname because it does not begin with a colon.

Double colons (::) in a pathname are used to specify the current directory's parent directory; triple colons specify the "grandparent" directory (two levels up), and so on.

A partial pathname that contains no colons is called a leafname. Command names, for example, are recognized by MPW as leafnames.

▶ Selecting Text with the § Character

MPW commands that take file names as parameters can also act on the current selection, or selected text, in a window. The § character (Option–6) can be used in a command to represent the currently selected text in a window. The § character can be used in two ways. You can use § standing by itself to mean "the currently selected text in the target window." Or you can use it as a extension to a window name, like this

```
name.§
```

to mean "the currently selected text in window *name*."

For example, you could use § with the Count command, which counts lines or characters in a file, in this fashion:

```
Count -l InsideMPW.§
```

to count the lines in the text currently selected in the InsideMPW window.

Variables in Pathnames

In MPW, you can specify pathnames using shell variables. For example, the {MPW} shell variable, defined in the MPW Startup file, expands to form the full pathname for the MPW folder when you use it in a command. Thus, the Directory command, described later in this chapter, could be entered as

```
Directory "{MPW}"Examples
```

To define and redefine variables, you can use the Set command, as described earlier in this chapter. To see the values of all currently defined variables, you can enter the Set command without any parameters.

Wildcards in File and Directory Commands

By using the wildcard characters ? and ≈ (Option-X), you can specify a number of files at once in a command. The ? character matches any single character except a colon or a Return. (To match a question mark, use ??.)

The ≈ character, mentioned previously in this chapter, matches any string of zero or more characters not including a colon or a Return. For example, the command

```
Files ≈.text
```

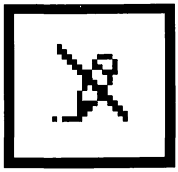
lists all file names in the current directory that end with the suffix ".text". For more on wildcard characters, see Chapter 4.

Locked and Read-Only Files

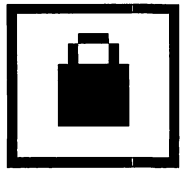
If you open a file that is locked, or is on a locked disk, the status panel in the upper left-hand corner of the Worksheet window displays a lock icon. When you see this icon, no editing or command execution is allowed in the locked window.

When you check out a read-only copy of a file from a project used by the MPW Projector tool, the file is always opened in read-only mode and a read-only icon is displayed in the status window.

MPW's read-only and locked icons are shown in Figure 2-13. Projector is described in the *MPW 3.0 Reference.*



The read-only icon



The locked icon

Figure 2-13. Read-only and locked icons

Examples of File and Directory Commands

In this section, we'll take a look at some of MPW's most often used file- and directory-related commands.

The Directory Command

The Directory command is used to set or list the current directory. Its syntax is:

```
Directory [-q | directory]
```

If you use the *directory* parameter, it becomes the default directory. The *directory* parameter can be a leafname; that is, a partial pathname that contains no colons. If this is the case, then MPW searches the {DirectoryPath} shell variable—a list of commonly used directives—for the *directory* parameter. If *directory* is a leafname and the {DirectoryPath} variable is undefined, MPW looks in the current directory for the directory you have specified.

If you do not use any parameters, Directory writes the full pathname of the current directory to standard output; this usually means that the pathname is printed on the screen.

If you don't use the -q option in a Directory command, pathnames that contain special characters are placed in quotation marks when Directory writes its list of directories to standard output. If you do use the -q option, pathnames are not quoted.

Directory returns a status code of zero if its operation is successful. A status code of 2 means that the directory was not found, that the command was aborted, or that there was a parameter error.

For example,

```
Directory HD:
```

sets the default (current) directory to the volume HD.

This variation:

```
Directory HD:InsideMPW:Chapter2:
```

sets the default directory to the Chapter2 folder in the InsideMPW folder on the volume HD. Another example:

```
Directory:InsideMPW:Chapter2:
```

sets the default directory to the folder Chapter2 in the folder InsideMPW in the current default directory.

Here is one last example:

```
Set DirectoryPath ":, {MPW}, {MPW}Examples:"
Directory CExamples
```

In the first of these two command lines, the MPW variable {DirectoryPath}—a list of common directories which the shell maintains to speed changing directories—is set to ":, {MPW}, {MPW}Examples: ". The second command line then sets the current directory to the CExamples directory. When the second command is executed, the shell first searches the current directory for the CExamples folder. Then it searches the {MPW} directory, and finally searches the {MPW}Examples directory. If there is no CExamples directory in your current directory, the current directory is set to {MPW}CExamples.

▶ The SetDirectory Command

You can also set the default directory with the SetDirectory command. Its syntax is

```
SetDirectory directory
```

When you use the SetDirectory command, you must specify a *directory* parameter. SetDirectory sets the default directory to the directory you have specified. If *directory* is on the MPW Directory menu, the shell adds it to the Directory menu as the last menu item.

One difference between the commands Directory (described previously) and SetDirectory is that the Directory command does not affect the MPW Directory menu, while SetDirectory does.

The SetDirectory command is usually invoked from the MPW menu rather than from a command line. So directory names that you pass to SetDirectory should not contain any of the following special characters, which have special meanings when they appear in menu items:

```
- ; ^ ! < / (
```

If you did issue a SetDirectory command from a command line, like this

```
SetDirectory {MPW}Examples:
```

SetDirectory would set the default directory to the Examples folder in the {MPW} directory and would add {MPW}Examples: to the Directory menu if it were not already there.

▶ The Files Command

You can list files and directories by using the Files command. The syntax of the Files command is

```
Files [option. . .] [name_]
```

The Files command can take multiple parameters. Each parameter is the name of a disk or a directory. When you execute the Files command, it lists the contents of each parameter you specify. If the parameter is a directory name, Files lists all of the directory's subdirectories, in alphabetical order, and then lists all of the files in the directory, also in alphabetical order. If the parameter is the name of a volume, Files writes the names of the directories and files in the volume in alphabetical order. You can use options to change the default behavior of the Files command.

If you don't specify a directory as a parameter when you list a directory as a parameter, Files lists the subdirectories and files in the current directory.

The options that can be passed to the Files command are as follows.

Option	*Meaning*
-c *creator*	List only files with this *creator*
-d	List only directories
-f	List full pathnames
-i	Treat all arguments as files
-l	Long format (type, creator, size, dates, etc.)
-m *columns*	Multicolumn column format, where *m* = *columns*
-n	Do not print header in long or extended format
-o	Omit directory headers
-q	Do not quote file names with special characters
-r	Recursively list subdirectories
-s	Suppress the listing of directories
-t *type*	List only files of this *type*
-x *format*	Extended format with the fields specified by *format*

The following characters can be used to specify the -x option's *format* parameter.

Character	*Meaning*
a	Flag attributes
b	Logical size, in bytes, of the data fork
r	Logical size, in bytes, of the resource fork
c	Creator of File ("Fldr" for folders)
d	Creation date
k	Physical size in kilobytes of both forks
m	Modification date
t	Type
o	Owner (only for folders on a file server)
g	Group (only for folders on a file server)
p	Privileges (only for folders on a file server)

The Files command returns a status code of 0 if all names are processed successfully. It returns a status code of 1 if there was a syntax error and a status code of 2 if any other kind of error occurred.

If you enter the command

```
Files -d
```

Files lists only the directories in the current directory, like this:

```
:Chapter1:
:Chapter2:
```

However, if you enter the command

```
Files -r -s -f
```

the result is a recursive list of the contents of the current directory. Full pathnames are used, and the printing of directory names is suppressed, as follows

```
'HD:Masterpiece:Backup of Chapter 1'
'HD:Masterpiece:Chapter 1'
'HD:Masterpiece:Ch1.Illustrations:1-01, Mac IIfx.MP'
'HD:Masterpiece:Ch1.Illustrations:1-02, Overlapping.MP'
'HD:Masterpiece:Ch1.Tables:box 1-1, mac evolution'
'HD:Masterpiece:Ch1.Tables:c headers.txt'
```

If you use the parameters -i, -x, k, and d with the Files command, you'll get a neat display listing the name, size, creation date, and creation time of any volume that you specify. For example, if the volume name of your hard disk is HD, and you issue the command

```
Files -i -x kd HD:
```

the Files command lists the size and creation date of the {CIncludes} directory:

```
Name          Size       Creation-Date
----------    -------    ----------------
HD:           37622K     2/2/91  4:35 PM
```

If you use the option -m, followed by the number of columns that you want printed, Files writes its list in multicolumn format. For example, the command

```
Files -m 2
```

prints a listing like this:

```
:Ch1.Illustrations:     'Backup of Chapter 1'
:Ch1.Tables:            'Chapter 1'
```

▶ The NewFolder Command

You can create new folders with the NewFolder command. Its format is

```
NewFolder name...
```

In this case, NewFolder creates new directories with the names specified in the *names* parameter. Any parent directories included in the name specification must already exist. The NewFolder command can be used only on hierarchical file system (HFS) disks, the current Macintosh standard.

For example, the command

```
NewFolder Tigers
```

creates a folder named Tigers as a subdirectory in the current directory.

▶ The Volumes Command

The Volumes command can provide you with information about all mounted disk volumes or about a single disk. You can use it to obtain the name of a volume, the names of all mounted volumes, or more detailed information about the volumes with which you are working.

This is the syntax of the Volumes command:

```
Volumes                        Displays information about
                               all mounted volumes.
Volumes [-l|-q] [volume...]    Displays information about
                               the volume specified.
```

If you use the -l option with the Volume command, it prints information in a "long" format, including not only each volume's name, but also its capacity, free space, number of files, and number of directories. If you do not use the -l option, Volume prints only the names of volumes.

When you use the -q option, Volume doesn't enclose names that contain spaces in quotation marks. If you don't use the -q option, names containing spaces are quoted.

The Duplicate Command

You can copy files or directories with the Duplicate command. Its syntax is

```
Duplicate [-y|-n|-c] [-d|-r] name... targetName
```

Duplicate copies file or directory *name* to file or directory *targetName*. For more information about its operation and its various options, see Appendix A.

The Catenate Command

With Catenate, you can merge multiple files into one file. Catenate also reads the data fork of a document and writes its contents to the output stream. One example of the use of Catenate was presented earlier in this chapter. More details are provided in Appendix A.

The Move Command

The Move command can be used to move files or directories from one directory to another. Its syntax is

```
Move [-y | -n | -c] [-p] name... target
```

Move takes two parameters, both of which can be file or directory names. The Move command moves *name* to *targetName*. If *targetName* is a directory, then *name* is moved into that directory. If *targetName* is a file or does not exist, then *name* replaces *targetName*, and the old *targetName* is deleted.

If a name is a directory, then its contents, including all of its subdirectories, are also moved.

Before it deletes a file or a directory, Move displays a confirmation dialog. You can override the dialog by using the option -y, -n, or -c.

The options that can be used with Move are as follows:

Option	*Meaning*
-y	Overwrite target files (avoids dialog).
-n	Do not overwrite target files (avoids dialog).
-c	Cancel if conflict occurs (avoids dialog).
-p	Write progress information to diagnostics.

For example,

```
Move -y File1 File2
```

moves File1 to File2, overwriting File2 if it exists. The result of the command is the same as renaming File1. Since the option -y is used, Move does not display a confirmation dialog.

The command

```
Move VirusDoc Capture {SystemFolder}
```

moves the files VirusDoc and Capture from the current directory to the system folder.

And the command

```
Move ThatKid ::
```

moves the file ThatKid from the current directory to the enclosing (parent) directory.

▶ The Rename Command

You can rename files and directories with the Rename command. Its syntax is

Rename [-y | -n | -c] *oldName newName*

When you execute the Rename command, the file, folder, or disk *name* is renamed *newName*. If the rename would overwrite an existing file or folder, a dialog box requests confirmation. You can override the dialog with a -y, -n, or -c option.

Rename cannot change the directory in which a file resides. To do that, you can use the Move command.

Rename accepts the following options.

Option	*Meaning*
-y	Overwrite existing file (avoids dialog).
-n	Do not overwrite existing file (avoids dialog).
-c	Cancel if conflict occurs (avoids dialog).

Consider the following examples. The command

```
Rename Untitled: Backup:
```

changes the name of the disk Untitled to Backup. The command

```
Rename HD:Programs:Prog.c Prog.Backup.c
```

changes the name of Prog.c in the HD:Programs directory to Prog.Backup.c in the same directory. The command

```
Rename File1 File2
```

changes the name of File1 to File2.

```
Rename -c File1 File2
```

changes the name of File1 to that of File2; if a conflict occurs, the operation is canceled.

▶ The SetFile Command

You can set the attributes of a file with the SetFile command. SetFile can be used to change a file's creator, file type, creation date, or modification date, or to specify whether a file is

- a system file
- bundled or unbundled
- locked or unlocked
- visible or invisible
- an "Init" file (a file executed when the system is booted)

- displayed on the desktop
- switch-launched (launched from another application), if possible
- shared (capable of being run multiple times)

The syntax of the SetFile command is

```
SetFile [option...] file...
```

The options that can be passed to SetFile are as follows.

Option	*Meaning*
-a *attribute*	Attributes (lowercase = 0, uppercase = 1)
-c *creator*	File creator
-d *date*	Creation date (*mm/dd/yy* [*hh:mm*[:*ss*] [AM | PM]])
-l h,v	Icon location (horizontal,vertical)
-m *date*	Modification date (*mm/dd/yy* [*hh:mm*[:*ss*] [AM | PM]])
-t *type*	File type

A period (.) can be passed as a *date* parameter to represent the current date and time.

Following the -a option, these letters can be used:

Letter	*Meaning*
A	Always switch launch (if possible)
B	Bundle
D	Desktop
I	Init file
L	Locked
M	Shared (can run multiple times)
S	System
V	Invisible

For example, the command

```
SetFile Balderdash -m "2/15/86 2:25"
```

sets the modification date of the Balderdash file. The command

```
SetFile Shrdlu -m .
```

sets the modification date of the Shrdlu file to the current date and time. In this example, the period is a parameter to the -m option, indicating the current date and time. The command

```
SetFile -c "MPS " -t MPST ResEqual
```

sets the creator and type for the MPW Pascal tool ResEqual.

▶ The Print Command

Print is the MPW command that prints source files, documents, and other kinds of text on a printer. If you select a block of text before executing Print, only the selected text will be printed.

The Print command writes its output to the currently selected printer. To use a printer, you must install the proper printer driver. You can then choose a printer using the Chooser desk accessory.

The syntax of the Print command is

```
Print [option...] file...
```

The options that can be used with Print are as follows.

Option	*Meaning*
-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).

Option	*Meaning*
-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*.
-ps *filename*	Include PostScript file as background for each page.
-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).

Status codes returned by the Print command are as follows.

Code	*Meaning*
0	Successful completion.
1	Parameter or option error.
2	Execution error.

The font normally used by Print is specified in a resource fork where the MPW editor stores font information. To print in a font other than this default font, you can use the -f option.

Consider the following examples of statements using the Print command. The command

```
Print §
```

prints the current selection. The command

```
Print -h -size 8 -ls 0.85 Source.c Source.r
```

prints the files Source.c and Source.r with page headers, using 8-point Monaco type and compressing line spacing. The command

```
Print -b -hf Courier -hs 12 -r Startup UserStartup
```

prints the Startup and UserStartup scripts in 12-point Courier type, with borders and headers, and with the pages in reverse order.

▶ Structured Constructs

This chapter concludes with an examination of the structured constructs used in the MPW command language.

▶ The If Command

The If command is used to create conditional loops. An If statement always begins with the command If, and ends with the word "End." The word "End" must always appear alone on a line. Any number of Else If statements may appear between the If command and the word "End."

The syntax of an If statement is

```
If expression
    command...
[Else If expression
    command... ] ...
[Else
    command... ]
End
```

(Note that Listing 2-2 showed how the If command is used in the MPW shell language.)

▶ The For Command

In the MPW command language, the For command does not work by iterating a numeric counter through a specified range of values. Instead, it repeats a set of commands for each parameter in a list. It can thus repeat an operation on a list of command parameters or on a list of file names.

When you use the For command, you must follow it with the name of a variable, the word "In," and a list of command parameters or filenames. On succeeding lines, you can specify commands to be carried out by your For loop. At the end of the loop, you must place a line containing on the word "End."

You must end each line with a Return character or a semicolon (;) command terminator.

The syntax of the For command is

```
For name In parameter...
   commands...
End
```

Status codes returned by the For command are as follows.

0 no *parameter* specified
–3 error in *parameter*

When you use a For loop in a script, MPW executes the list of commands once for each word from the "In *parameter*..." list. The current *parameter* is assigned to the variable *name*, and you can therefore refer to it in the list of *commands* in the For loop by using the notation {*name*}.

You can terminate a For loop with a Break command, and you can terminate the current iteration of the loop with a Continue command.

After the word End that terminates a For loop, you can use the command terminators |, &&, and | |, described earlier in this chapter. You can redirect the output of a For loop by using the redirection operators <, >, >>, ≥, ≥≥, Σ, and ΣΣ following the word End. If you use these optional command terminators and redirection operators, they apply to the entire For loop.

One way to use a For loop is:

```
For n In 1 2 3
   Echo n = {n}
End
```

The following example echoes the following list to standard output, normally the screen:

```
n = 1
n = 2
n = 3
```

A more useful example is as follows.

```
For FileName In ≈.p
   Pascal "{FileName}"
   Echo '"{FileName}" compiled.'
End
```

This example compiles every Pascal file in the current directory; that is, every file with a name that ends with the suffix ".p". During the loop, it echoes to standard output the name of each file that has been compiled.

The following For loop

```
For Filename In ≈.c
   Rename -y temp "{Filename}"
   Print -h "{Filename}"
   Echo "{Filename}"
End
```

prints all C source files in the current directory. During the loop, the Print command prints each file with a heading, and the Echo command echoes the name of each file printed.

▶ The Loop Command

Loop is a command that is used to set up a structured loop. The loop starts with the Loop command and ends with an End command. Any commands between Loop and End repeat indefinitely, or until the loop is terminated with a Break command. Within the loop, the Continue command can be used to terminate the current iteration.

The syntax of the Loop command is

```
Loop
   command...
   [Break]
End
```

When a Loop command terminates, it returns the status of the last command executed in the shell variable {Status}.

Following is a short script that uses Loop to execute a command several times, once for each parameter passed to it as a parameter variable.

```
Set parameter {1}
Loop
   Shift
   Break If {1} ==
   {parameter} {1}
End
```

The Shift command is used to step through the parameters, and the Break command ends the loop when all parameters passed to the script have been exhausted.

The Break Command

The Break command, as shown in the above examples, is used to exit from a For or Loop command. Its syntax is

```
Break [If expression]
```

Status codes returned by the Break command are

Code	*Meaning*
0	no errors detected
–3	Break was used outside a For . . .End or Loop . . . End construct, or the parameters passed to Break were incorrect
–5	invalid expression

If there is no *expression* parameter, or if the Break command's *expression* parameter is nonzero, Break terminates execution of the For or Loop construct in which it is most closely nested.

The following loop shows how Break is used with an *expression* parameter.

```
Set Exit 0
   For file in ≈.c
   Break If {Status} != 0
   Rename -y temp "{FileName}"
   Print -h "{FileName}"
   Echo "{FileName}"
End
```

This loop, like the second example given for the Loop command, prints all C source files in the current directory. However, in this case, Break terminates the loop if a nonzero status value is returned.

▶ The Continue Command

The Continue command can be used to terminate an iteration in a For or Loop command and to continue with the next iteration.

This is the syntax of the Continue command:

```
Continue [If expression]
```

If there is no *expression* parameter, or if the *expression* parameter of a Continue command is nonzero, Continue terminates the current iteration of the For or Loop construct in which it is most closely nested, and continues with the next iteration. If no further iterations are possible, the loop is terminated.

▶ A Modified Startup Script

Listing 2-4 is a Startup script that has been edited to use the modified directory structure explained in this chapter. Also, some shell variables have been changed to suit my preferences. If you would like to modify your own Startup script, you can use Listing 2-4 as a guide.

Listing 2-4. Modified UserStartup script

```
# Modified Startup Script
# By Mark Andrews
#
# (Original provided by MPW)

# {Boot} - Boot disk (Predefined)

      Export Boot

# {SystemFolder} - Directory that contains
# the System and Finder (Predefined)

      Export SystemFolder

# {ShellDirectory} - Directory that contains
# the MPW Shell (Predefined)

      Export ShellDirectory

# {Active} - Active (topmost) window (Predefined)

      Export Active

# {Target} - Target (previously active)
# window (Predefined)

      Export Target

# {Worksheet} - Name of the Worksheet window (Predefined)

      Export Worksheet

# {Status} - Result of last command executed (Predefined)

      Export Status

# {User} - Automatically defined to the
# name the appears in the Chooser (Predefined)

      Export User
```

Listing 2-4. Modified UserStartup script (continued)

```
# {MPW} - Volume or folder containing MPW

      Set MPW "{ShellDirectory}"
      Export MPW

# {Commands} - Directories to search for commands

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

# {Libraries} - Directory that contains shared libraries

      Set Libraries "{MPW}Libraries:Libraries:"
      Export Libraries

### The following variables have been modified
### {AFolder}, {CFolder}, {PFolder} and {RFolder} added

# {AIncludes} - Directories to search for
# assembly language include files

      Set AIncludes "{MPW}AFolder:AIncludes:"
      Export AIncludes

# {CIncludes} - Directories to search for C include files

      Set CIncludes "{MPW}CFolder:CIncludes:"
      Export CIncludes

# {CLibraries} - Directory that contains C libraries

      Set CLibraries "{MPW}CFolder:CLibraries:"
      Export CLibraries

# {PInterfaces} - Directories to search
# for Pascal interface files

      Set PInterfaces "{MPW}PFolder:PInterfaces:"
      Export PInterfaces
```

Listing 2-4. Modified UserStartup script (continued)

```
# {PLibraries} - Directory that contains Pascal libraries

      Set PLibraries "{MPW}PFolder:PLibraries:"
      Export PLibraries

# {RIncludes} - Directory that contains Rez include files

      Set RIncludes "{MPW}RFolder:RIncludes:"
      Export RIncludes

### Modified variables end here

# {CaseSensitive} - If nonzero,
# pattern-matching is case sensitive

      Set CaseSensitive 0
      Export CaseSensitive

# {SearchBackward} - If nonzero, search goes backwards

      Set SearchBackward 0
      Export SearchBackward

# {SearchWrap} - If nonzero, search wraps

      Set SearchWrap 0
      Export SearchWrap

# {SearchType} - Specifies the default searching type
# (0/literal, 1/word, 2/regular expression)

      Set SearchType 0
      Export SearchType

### {Tab}, {Font} and {FontSize} have been modified

# {Tab} - Default tab setting for new windows

      Set Tab 5                # Default is 4
      Export Tab
```

Listing 2-4. Modified UserStartup script (continued)

```
# {Font} - Default font for new windows

      Set Font Courier      # Default is Monaco
      Export Font

# {FontSize} - Default font size for new windows

      Set FontSize 10       # Default is 9
      Export FontSize

### Modified variables end here

# {AutoIndent} - If nonzero, auto indentation
# is the default for new windows

      Set AutoIndent 1
      Export AutoIndent

# {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 nonzero, command files
# terminate after the first error

      Set Exit 1
      Export Exit

# {Echo} - If nonzero, echo commands before execution

      Set Echo 0
      Export Echo
```

Listing 2-4. Modified UserStartup script (continued)

```
# {Test} - If nonzero, don't execute tools and applications

      Set Test 0
      Export Test

# {Windows} - A list of all open windows (predefined)

      Export Windows

# {Aliases} - A list of all open windows (predefined)

      Export Aliases

# {Commando} - Name of the Commando dialog (predefined)

      Set Commando Commando
      Export Commando

# Alias definition

      Alias File Target

# Execute UserStartup script

      Execute "{ShellDirectory}UserStartup"

#     Execute supplementary UserStartup scripts
#     (files with names like UserStartup•MyStartup)

      For __Startup__i in `(Files ∂
      "{ShellDirectory}"UserStartup•≈ ∂
      || Set Status 0) ≥ dev:null`
            Execute "{__Startup__i}"
      End
      Unset __Startup__i
```

▶ Conclusion

This chapter described the most important features of the MPW command language and explained how to write MPW commands and MPW scripts. It also explained how MPW manages files and directories and how to write commands and scripts that affect files and directories.

3 ▶ Menus and Dialogs

What do you get when you cross a line-oriented command language such as UNIX with a window-based, mouse-driven Macintosh application program? That was the question that Apple's engineers faced when they sat down to design MPW. What they wanted was a software development system that would combine the best features of a UNIX-like command-line interpreter with the convenience of a well-behaved window-based Macintosh-style application.

The way they finally solved the problem was to wrap both kinds of programming environments into a single package. First, they designed a command language, a command-line interpreter, and a text editor that would enable the user to execute commands from scripts and command lines. Then they added a Macintosh-style interface that also made it possible to execute commands using windows, pull-down menus, and click-and-close dialog boxes.

In Chapter 2, you learned how to write command lines and scripts, and how to execute commands from scripts and command lines. In this chapter, you will learn how to

- issue MPW commands by selecting items from pull-down menus
- run scripts and applications by selecting pull-down menu items
- add your own menus and items to the MPW menu bar
- modify your UserStartup script so that your customized menus and menu items are initialized every time you start up MPW
- create dialog boxes that can be used in MPW scripts

- execute commands using a special kind of dialog box called a Commando dialog
- change the appearance of a Commando dialog using MPW's built-in Commando editor

The MPW Menu Structure

Every MPW command that can be issued from a command line can also be executed by selecting its name from a pull-down menu. Of course the MPW menu bar does not contain an item for every command; it is not nearly big enough to hold them all. But if there is a command that you use often, and you find that it does not appear anywhere on the MPW menu tree, you can easily put it there by either adding its name to an existing menu or placing it under a new menu that you have added to the menu bar.

Furthermore, when you want to add a command to the MPW menu structure, it does not have to be an existing MPW command. If you have written a script or a tool that you would like to add to the MPW menu structure, you can create a menu item for it, and then select it from a pull-down menu, in the same way that you would select any other menu item.

If you have created a customized menu structure and want it to appear every time you launch MPW, you can instruct MPW to build your menu every time it starts up by adding commands to build the menu to your UserStartup script.

Procedures for customizing the MPW menu bar are explained later in this chapter. First, though, let's take a look at the menu bar and its menus and menu items, and see what they all do. Figure 3-1 shows the MPW menu structure.

What's on the Menu

Not counting the Apple menu, the following eight menus appear on the MPW 3.2 menu bar. The items listed under each menu will be covered individually later in this chapter.

Menu Name	*Function*
File	Used to create, open, print, close, and save files.
Edit	Contains items that you can select to edit text. In addition to the usual Macintosh editing commands, the MPW Edit menu contains several special items.

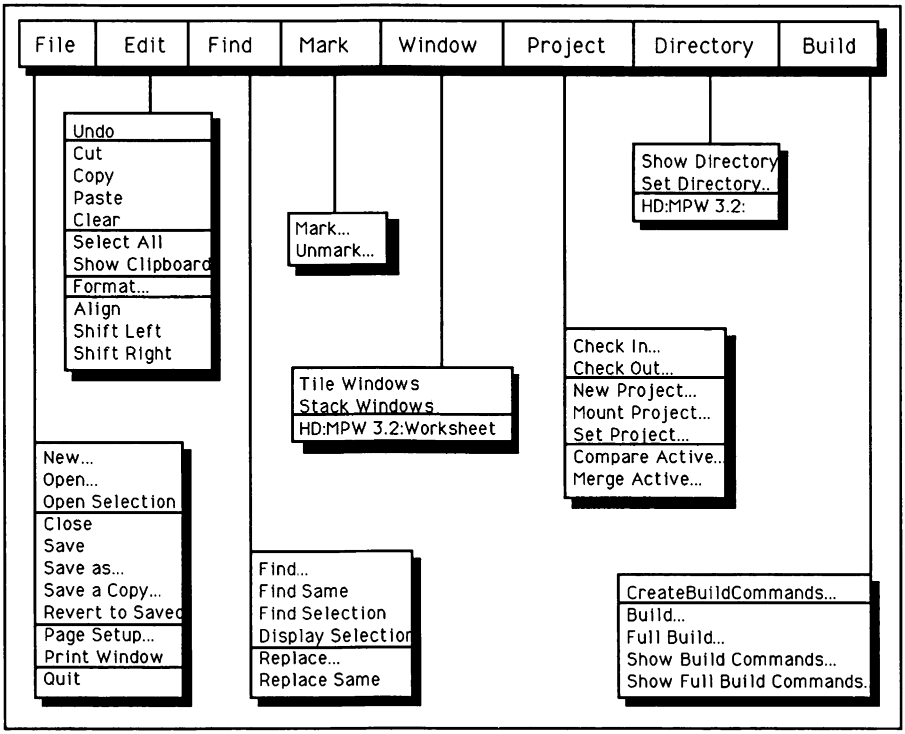


Figure 3-1. The MPW menu structure

Menu Name	*Function*
Find	Lists commands for finding and replacing text.
Mark	Provides a method for marking locations in a document so that they can be found quickly during viewing and editing operations.
Window	Used to bring a window to the front; lists the names of all currently open windows.
Project	Used to control an MPW project management tool called Projector.
Directory	Used to obtain the name of the current directory, or to change the current (default) directory.
Build	Used to build programs by converting raw source code into object-code modules that can be sent to the MPW Linker to be converted into executable programs.

The File Menu

When you select the File menu, shown in Figure 3-2, it presents a list of items that can carry out shell commands for creating, opening, printing, closing and saving files.

New

You can issue the MPW command New by selecting the menu item "New" under the File menu. You can select the New menu item by either choosing it with the mouse or pressing its keyboard equivalent, Command-N. You can also issue the New command from a command line or a script, as explained in Chapter 2. When you select New, it displays a dialog window like the one illustrated in Figure 3-3.

The New dialog window is a Standard File Manager dialog box that you can use to select the directory location of a new document, type in the new document's file name, and create the new document. The New dialog contains buttons for changing disk drives, ejecting a disk, creating the new document and closing the dialog, and canceling your New command—that is, closing the dialog without creating a new document.

When you have created a document using the menu selection New, the New dialog disappears and an empty window in which you can enter the new document's text appears on the screen.

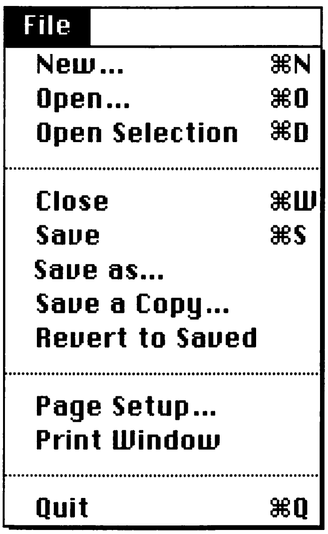


Figure 3-2. File menu

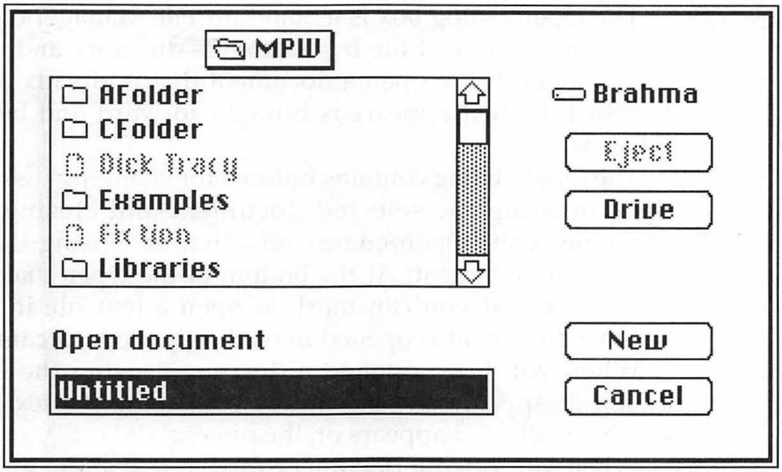


Figure 3-3. New dialog box

Open

When you select the Open item under the File menu, it displays a dialog window like the one shown in Figure 3-4. You can then use the dialog to open any file. You can also open a file by executing the MPW command Open.

To select the Open menu item, you can click on it with the mouse or press its command-key equivalent, Command-O.

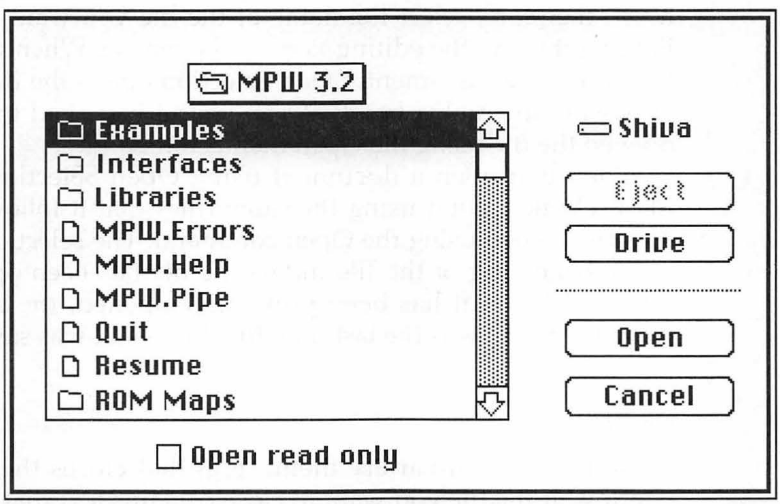


Figure 3-4. Open dialog box

The Open dialog box is a Standard File Manager dialog that you can use to open any text file by finding its directory and clicking on its file name. If you try to open a document that is already open, the window in which its text appears is brought forward and becomes the active window.

The Open dialog contains buttons for changing disk drives, ejecting a disk, opening the selected document and closing the dialog, and canceling your Open command—that is, closing the dialog without opening a document. At the bottom of the Open dialog, there is also a check box that you can mark to open a text file in read-only mode. When a document is opened in read-only mode, it cannot be edited.

When you have opened a document using the Open dialog, the dialog disappears and a window containing the text of the document you have selected appears on the screen.

When you open a document for the first time, its selection point—that is, the cursor location—is placed at the beginning of the document. When you close a document, however, MPW saves its selection point, and that is where the selection point is placed the next time the document is opened. The MPW Editor automatically scrolls the document to make the selection point visible on the screen.

Open Selection

Before you choose the Open Selection menu item, the name of the file you want to select must appear somewhere on the screen, for example, in a list of documents returned by a file or directory command. You must, therefore, select the name of the file you want to open by highlighting it using the editing keys or the mouse. When you have selected the name of a document, Open Selection opens the document for you, bypassing the dialog box that you would have had to used if you had opened the file using the Open menu command.

When you open a document using Open Selection, MPW sets the file's selection point using the same rules that it follows when a document is opened using the Open command. The selection point is placed at the beginning of the file unless the file has been opened previously under MPW. If it has been previously opened, the selection point is placed where it was the last time the document was saved.

Close

Close is a garden-variety menu item that closes the currently active window. If the file you want to close contains text that has been altered, Close displays a dialog window asking you if you want to close the document without saving its changes.

The command-key equivalent of Close is Command-W. You can also issue a Close command from a command line or a script, as explained in Chapter 2.

Save

The Save menu item saves the contents of the currently active window. When you are writing or editing a document, it is a good idea to save its contents frequently, so you won't lose hours of work if you hit a few keys incorrectly or become the victim of an equipment malfunction.

The keyboard shortcut for Save is Command-S. As explained in Chapter 2, you can also issue the Save command from a command line or a script.

Save As

The Save As menu item displays a dialog box that allows you to save the contents of the currently active window as a new file in a directory that is different from the current directory, or under another name. When you have saved a file using Save As, the name of the new file replaces the name of the original file in the current window's title bar, and all subsequent editing changes affect the new file, not the old one.

Save a Copy

The Save a Copy command works much like Save As. It also displays a dialog box that allows you to save the contents of the currently active window as a new file in a different directory or under another name. However, when you have saved the file, the name of the original file remains in the active window's title bar, and all subsequent editing changes still affect the original file rather than the new one.

Revert to Saved

The Revert to Saved menu item discards any changes that you have made to the document in the active window since the last time you saved it. It thus restores the last saved version of the file.

If you have not made any modifications in the currently active window since the last time it was saved, the Revert to Save item is dimmed and cannot be selected.

Page Setup

The Page Setup menu item displays the Printing Manager's standard Page Setup dialog, which allows you set up printing parameters to match your printer and your preferences.

Print Window and Print Selection

If no text is selected in the active window, the item that follows Page Setup on the file menu reads Print Window. However, if you select a block of text in the active window using the editing keys or the mouse, the name of the Print Window item changes to Print Selection.

When no text is selected, choosing the Print Window menu item prints the full text of the document displayed in the active window. Of course, your printer and the appropriate print drivers must be installed correctly for Print Window to work correctly.

When you have selected a block of text and the name of the Print Window item has changed to Print Selection, only the text that has been selected is printed.

MPW does not have a very sophisticated set of print formating capabilities and does not display a dialog box asking for printing preferences when you select Print Window. But you can set some printing parameters by modifying the {PrintOptions} shell variable in your MPW startup script. By resetting the options of the {PrintOptions} variable, you can change

- the number of copies to be printed
- which pages to print
- print quality (on an ImageWriter printer)
- the font used for printing
- the type size used for printing
- page headings
- the title of a document
- margin settings
- whether pages are printed in consecutive or reverse order

The options that you can modify by changing the {PrintOptions} variable are described in Chapter 2.

Once the {PrintOptions} variable has been changed, its new settings remain in effect with every print job until they are changed again. One way to use different printing options for different documents might be to modify your MPW menu structure by adding a Print Options menu. An easier solution might be to print your MPW documents using a word processor.

Quit

The Quit menu item exits MPW. If there are open files that have been modified since the last time they were saved, Quit displays a dialog for each modified file, asking you if you want to save the changes you have made in it before you exit MPW.

The keyboard equivalent for the Quit menu is Command-Q. The Quit command can also be issued from a command line or a script. For descriptions and examples of the Quit command and other commands that have menu equivalents, refer to Appendix A.

▶ The Edit Menu

The Edit menu, shown in Figure 3-5, offers the standard Macintosh Cut, Paste, Copy, and Clear menu items, along with several special items that you won't find under the Edit menus provided by most applications. The nonstandard items provided by MPW are Align, Shift Left, and Shift Right.

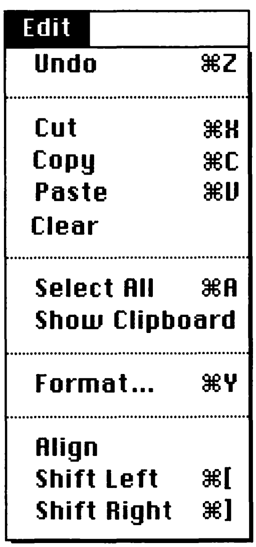


Figure 3-5. Edit menu

Undo

The Undo menu item discards the effects of the most recent modification to the text in the active window. It does not undo changes to resources such as font or tab settings. The command-key equivalent for Undo is Command-Z. The Undo command can also be issued from a command line or a script.

Cut

The Cut menu item copies text that has been selected to the Clipboard and then deletes it from the document being edited. The command-key equivalent of the Cut item is Command-X. The Cut command can also be issued from a command line or a script.

Copy

The Copy menu item copies text that has been selected to the Clipboard, without deleting it from the document being edited. The command-key equivalent of the Copy item is Command-C. The Copy command can also be executed from a command line or a script.

Clear

The Clear menu item deletes any text that has been selected in the currently active window without copying it to the clipboard. The keyboard equivalent for Clear is the Clear key. Clear command can also be issued as a command from a command line or a script.

Paste

The Paste menu item inserts any text that is currently on the Clipboard into the currently active window, beginning at the selection point. The keyboard equivalent for Paste is Option-V. Paste can also be issued as a command from a command line or a script.

Select All

The Select All menu item selects all the text in the document displayed in the active window. Its command-key equivalent is Command-A.

Show Clipboard

If any text has been copied to the Clipboard, the Show Clipboard menu item displays it.

Format

The Format menu item displays a dialog that you can use to change the font and type size of the text displayed in the active window. MPW's Format item, unlike the Type and Style items on the menus of most word processors, alters the font and size of all the text in the document being edited, not just text that has been selected.

▶ The Format Dialog

The Format dialog box is shown in Figure 3-6. In addition to its font and text-size controls, it has several other controls that can also be used to change the appearance of the document in the active window.

The Show Invisibles Check Box

The Show Invisibles check box can be selected to display nonprinting characters, that is, characters that are not ordinarily visible. These characters are shown in Table 3-1.

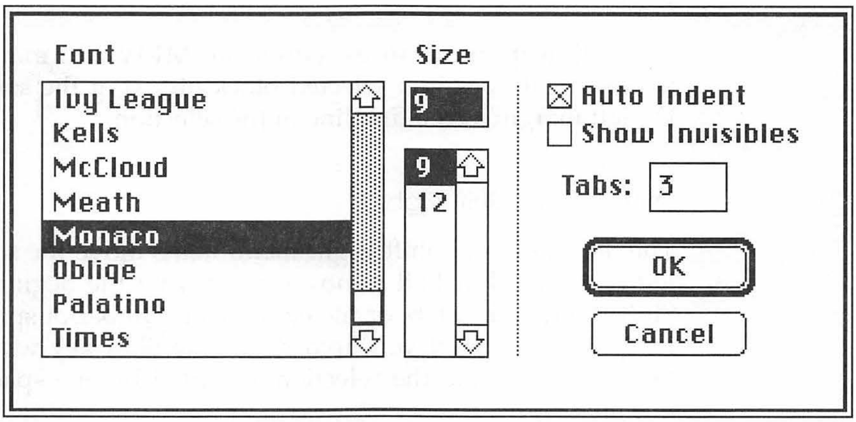


Figure 3-6. Format dialog box

Table 3-1. Nonprinting characters

Non-Printing Character	*Character Displayed*
Tab	Δ
Space	◊
Return	¬
All others	¿

When you want to delete nonprinting characters from a document, you can make them visible by checking the Show Invisibles check box. Its default setting is off.

The Tabs Check Box

The Tabs check box sets the number of spaces that are skipped when the Tab key is pressed. The default setting is four spaces.

Alternatives to Using the Format Dialog

You can also alter the appearance of the text in MPW documents by modifying the {Font}, {FontSize}, {Tab}, and {AutoIndent} variables in the MPW Startup script. Procedures for modifying the Startup script are explained in Chapter 2.

Align

The Align menu item executes the MPW command Align, which positions all lines in a selected block of text at the same distance from the left margin as the first line in the selection.

Shift Left and Shift Right

The Shift Left and Shift Right menu items move the selected text to the left or right. Shift Left removes a tab from the beginning of each line. Shift Right adds a tab, or the equivalent number of spaces, to the beginning of each line. If you hold down the Shift key while selecting Shift Left or Shift Right, the selection is shifted by one space rather than by one tab stop.

The keyboard equivalent of Shift Left is Command-[. The keyboard equivalent of Shift Right is Command-].

▶ The Find Menu

Find, shown in Figure 3-7, is the menu to use when you want to find, or find and replace, strings of text in a document. With the items listed under the Find menu, you can perform some pretty complex searching and search-and-replace operations. But you can perform string-matching and pattern-matching operations that are even more powerful by executing commands from command lines and scripts, as explained in Chapter 4.

Find

The menu item Find—the first item under the menu with the same name—can find any block of text in an MPW document. The command-key equivalent for Find is Command-F. The Find command can also be issued from a command line or a script, as described in Chapter 4.

Normally, the Find command begins its search at the location of the insertion point, or text cursor, and proceeds toward the end of the document displayed in the active window. However, if you hold down the Shift key as you select the Find menu item—or if you hold it down as you click the OK button of any dialog that Find displays—the search that you request is carried out in reverse. That is, it starts at the current location of the cursor and moves backwards, toward the beginning of the document being edited.

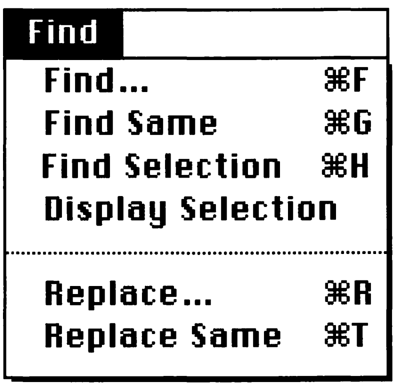


Figure 3-7. Find menu

By the Way ▶

The Search Command. In addition to the Find command, the MPW command language includes a Search command that can search through a list of files for any text pattern. More information about Search and Find can be found in Chapter 4.

▶ The Find Dialog

When you select Find, it displays a dialog box like the one in Figure 3-8.

The Find dialog contains a TextEdit box and ten button controls. Above the TextEdit box is the prompt, "Find what string?" In the TextEdit box, you can type the string that you want to find. You can then use the button items to specify exactly how you want your search carried out.

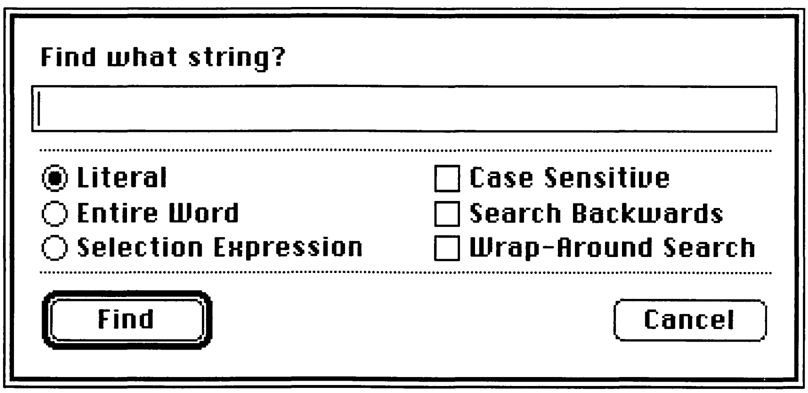


Figure 3-8. Find dialog

Radio Buttons

There are three radio buttons in the Find dialog box. They are arranged in a group, so only one of them can be selected. The labels and functions of these buttons are as follows.

- **Literal**: Click on this button, and Find searches for the exact string you have specified, anywhere it may appear, even if it is a part of other words or expressions.
- **Entire Word**: If you select this button, Find looks for the specified string only when it occurs in a document as a single word. The rules used for defining a word depend on the setting of the {WordSet} shell variable. In the MPW Startup script, the {WordSet} variable is defined this way:

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

  According to this definition, a word composed of any combination of the uppercase letters *A* through *Z*, the lowercase letters *a* through *z*, the numerals 0 through 9, and the underscore character (_). When {WordSet} is defined in this fashion, double-clicking on a word highlights all the alphabetic and numeric characters that it may contain. If the word contains an underscore character, that is also considered to be a part of the word. But if the word contains any special character such as a period or a colon, MPW stops highlighting the word at that point and does not select the special character.

  If you want to use double-clicking to select pathnames, you may want to change this definition by redefining the {WordSet} shell variable. The procedure for doing that is explained in Chapter 2.

- **Selection Expression**: When you highlight the Selection Expression button, the "Find what string?" prompt changes to "Find what selection expression?" You can then instruct MPW to search for a text pattern using a regular expression—a string made up of text characters and special characters that stand for text patterns in search-and-replace operations. Special characters that have special meanings in regular expressions are called regular expression operators.

A detailed discussion of regular expression operators is presented in Chapter 4. Table 3-2 lists a few regular expression operators that are used often in find-and-replace operations.

Table 3-2. Regular expression operators

Operator	*Description*	*Meaning*
/ . . . /	Slashes	Delineators used to enclose a regular expression, searching forwards.
\. . . \	Backslashes	Delineators used to enclose a regular expression, searching backwards.
?	Question mark	A wildcard character used to represent a single character.
≈	Option-X	A wildcard that stands for any string of zero or more characters that does not include a Return.
[*characterList*]	Square brackets	Used to enclose a list of characters. MPW searches for all characters specified in the list.
[¬*characterList*]	Square brackets enclosing a character list preceded by the symbol ¬ (Option-L)	MPW searches for all characters *not* specified in the list.

Consider the following example of a regular expression that could be used in a Find command. If you typed

```
/≈ Smith/
```

as a selection expression, MPW would search the document in the active window for the full name of any person whose last name was Smith.

You can also use any number as a selection expression. For example, if you typed the number

```
15
```

in the selection-expression window, the MPW Editor would move to the fifteenth line of the document in the target window and select that line.

Check Boxes

There are three check boxes in the Find dialog, and they can be selected in any combination. Their labels and functions are as follows.

- **Case Sensitive**: Normally, searches in MPW are not case sensitive. Checking this box specifies case-sensitive searching.
- **Search Backwards**: Normally, the Find command searches documents in a forward direction, beginning at the location of the insertion point and proceeding to the end of the document. Checking this box instructs MPW to conduct a backwards search, starting at the insertion point and moving in reverse toward the beginning of the document being edited.
- **Wrap-Around Search**: Unless the Search Backwards box has also been selected, checking this item instructs MPW to conduct a wrap-around search; that is, to search forward to the end of a document, and then to wrap around and search from the beginning of the document to the location of the cursor. If the Search Backwards box has also been checked, the direction of the wrap-around search is reversed.

Plain Buttons

The Find dialog has two plain buttons: one labeled Find and one labeled Cancel. If you click the Find button, MPW searches for the next occurrence of the selected string. Clicking Cancel cancels the Find operation.

The check boxes in the Find dialog set the shell variables {CaseSensitive}, {SearchBackward}, and {SearchType}. You can also set these variables by issuing commands from command lines or scripts. The procedures for setting variables from command lines and scripts are explained in Chapter 2.

Find Same

When you select the Find Same menu item, MPW repeats its last Find operation. You can also issue a Find Same command by typing Command-G.

Find Selection

The Find Selection menu item finds the next occurrence of the current selection. Its command equivalent is Command-H.

Display Selection

The Display Selection menu item scrolls into view the current selection in the active window.

Replace

You can perform search-and-replace operations in an MPW file by selecting the Replace menu item. You can also issue a Replace command by typing Command-R. When you select Replace, MPW displays a dialog window like the one in Figure 3-9.

The Replace dialog is similar to the Find dialog described earlier, but there are a couple of differences:

- The dialog displayed by Replace has two TextEdit windows: one labeled "Find what string?" and the other labeled "Replace with what string?" In the first TextEdit window, you can type the string that you want MPW to find. In the second, you can type a replacement string.
- The Replace dialog has four plain buttons. They are labeled Replace, Replace All, Find, and Cancel. The Replace button finds the next occurrence of the string in the edit box labeled "Find what string?" and replaces it with the string in the edit box labeled "Replace with what string?" The Replace All button replaces all occurrences of the string in the first edit box with the string in the second edit box.

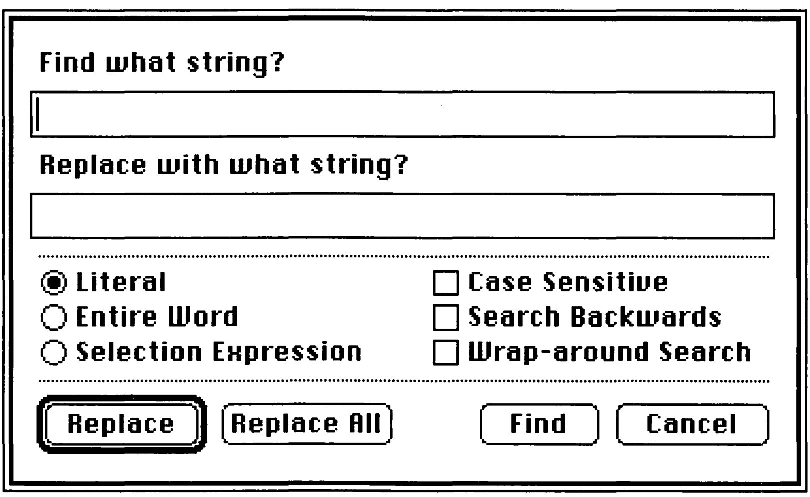


Figure 3-9. Replace dialog

Except for these differences, the Replace dialog carries out search-and-replace operations in exactly the same way that the Find dialog carries out search operations.

Replace Same

Replace Same repeats the last Replace operation. Its keyboard equivalent is Command-T.

▶ The Mark Menu

By using the Mark menu (Figure 3-10), you can place invisible markers in any text document and move your cursor to any marked item at any time at the click of a pull-down menu. And when you save a document, MPW also saves its markers, so they will be back again for you to use in your next editing session.

When there are no markers in the document displayed in the active window, the Mark menu has just two items: Mark and Unmark. When the document in the active window contains markers, the selections that have been marked are listed as additional menu items. A horizontal line separates them from the items Mark and Unmark, and they are arranged in the order in which they appear in the document being edited.

When you launch MPW 3.2 for the first time, you may notice that a number of MPW commands are listed as marked items. You can use these predefined markers to jump to the sections of the MPW Worksheet in which the marked commands are explained. Or, if you prefer, you can unmark the commands.

Mark

It is easy to place a mark in a document using the Mark menu. Just select any word or phrase that you want to mark, and pull down and select the Mark menu item. MPW then displays a dialog containing a

Figure 3-10. Mark menu

TextEdit window and the question, "Mark the selection with what name?" The dialog displayed by the Mark menu item is shown in Figure 3-11.

When the Mark dialog appears, its TextEdit window contains the string you have selected. But you can change the text to read any way you choose.

When you have decided what you want the name of your marker to be, you can click the dialog's OK button, and your marker is saved under the name you have chosen. You can also click the Cancel button to cancel your marking operation.

Unmark

When you want to delete a marker from a document, you can select the Unmark menu item. MPW then displays a dialog that lists all the current markers. The Unmark dialog is shown in Figure 3-12.

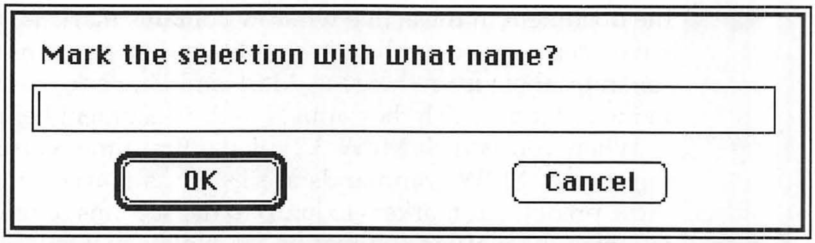


Figure 3-11. Mark dialog

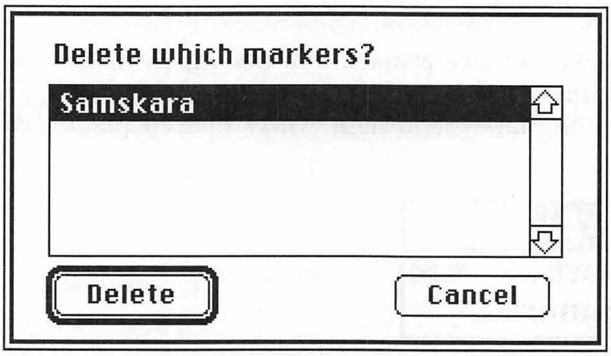


Figure 3-12. Unmark dialog

From the list of markers in the Unmark dialog, you can select one or more markers that you want to delete by clicking, or clicking and dragging, your mouse. You can then click the dialog's Delete button to delete all markers you have selected. You could also click the Cancel button to cancel your unmarking operation.

Jumping to a Marked Item

Once you have marked a location in a document, you can move the insertion point (text cursor) there instantly by pulling down the Mark menu and selecting the marker's name.

▶ The Window Menu

The Window menu, illustrated in Figure 3-13, has two functions. It lists all the currently open windows, and it can be used to control the arrangement of the windows on your screen.

When you first launch MPW, and the Worksheet window is the only window displayed, the Window menu has three items. The first item is labeled Tile Windows; the second is labeled Stack Windows. The third item—separated from the first two by a horizontal line—contains the full pathnames of all the open windows.

Each time you open a window, MPW adds its pathname to the Window menu's window list. Each time you close a window, its pathname is deleted. So the names of all the open windows are always displayed under the Window menu.

In the Window menu's list of open windows, a code is used to specify the status of certain windows. A check mark (√) precedes the name of the currently active window. A bullet (•) precedes the name of the target window, or second topmost window. If the name of a window is underlined, it means that the window has been modified since the last time it was saved.

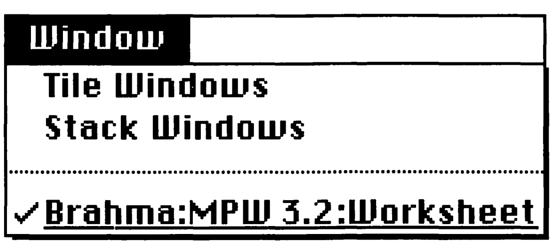


Figure 3-13. Window menu

When you choose a window name that is displayed under the Window menu, MPW brings the selected window to the front and makes it the active window.

For more information about how MPW manages windows, see Chapters 1 and 2.

Tile Windows and Stack Windows

The Stack Windows menu item arranges the windows on the screen in a stacked pattern, so that only the title bars of inactive windows are visible. Figure 3-14 shows a stacked window pattern.

When you select the Tile Windows menu item, MPW arranges all the open document windows in a tiled pattern on your screen. If necessary, Tile Windows reduces the sizes of the windows that are open to fit the pattern in which they are displayed. If there are two open windows, for example, Tile Windows splits the screen in half horizontally, and displays one window above the other.

If there are three open windows, Tile Windows divides the screen into horizontal thirds, as shown in Figure 3-15. If there are four windows, Tile Windows arranges them in a checkerboard pattern, as illustrated in Figure 3-16. When there are more than four windows, Tile Windows creates a checkerboard pattern with a smaller square for each window displayed.

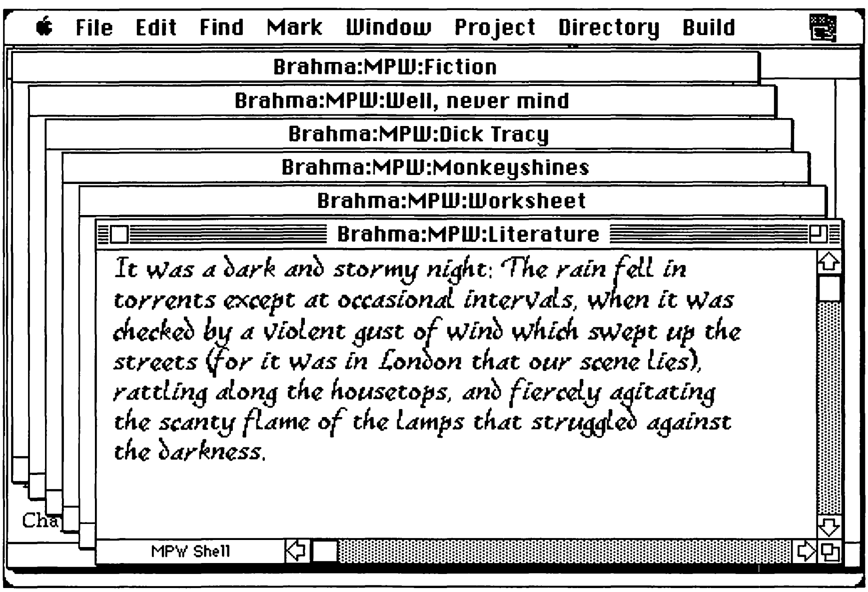


Figure 3-14. Stacked windows

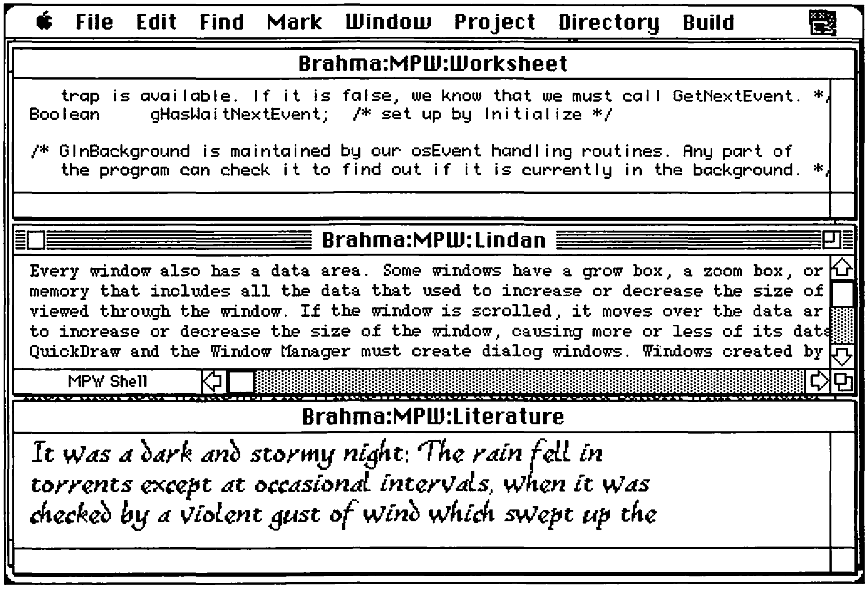


Figure 3-15. Tiled windows (horizontal)

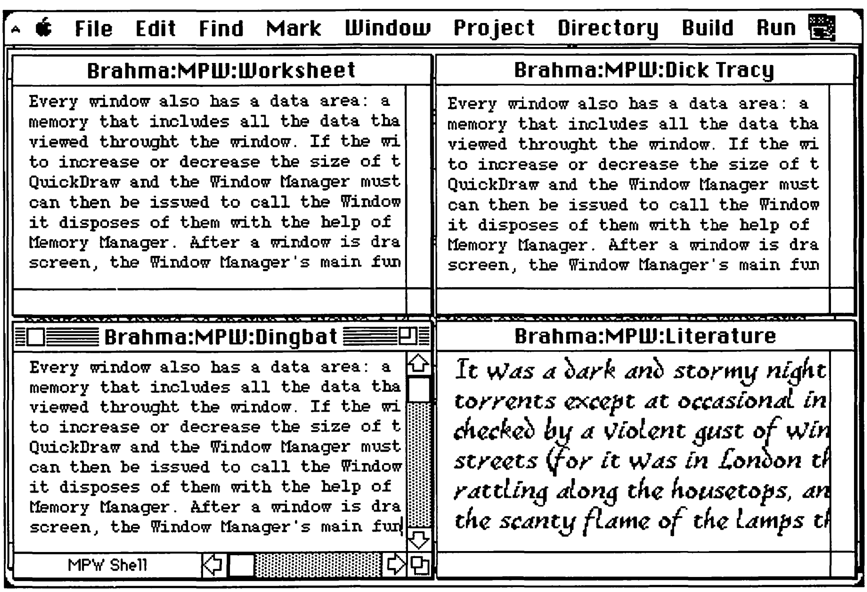


Figure 3-16. Tiled windows (checkerboard)

By the Way ▶

Tile Windows and the Worksheet Window. Normally Tile Windows does not include the Worksheet window in its tiled display. If you want the Worksheet window included, you must press the Option key while you select the Tile Windows menu item, or change the value of the {TileOptions} shell variable, as explained later in this chapter.

The Tile Windows and Stack Windows menu items create their displays by issuing the TileWindows and StackWindows shell commands. As mentioned in Chapter 2, you can also execute the TileWindows and StackWindows commands from a command line or a script.

If you issue a TileWindows command with an -h option, MPW tiles your windows in a horizontal arrangement. If you use a -v option, your windows are tiled vertically. For example, the command

```
TileWindows -h HD:InsideMPW:Chapter1
HD:InsideMPW:Chapter2
```

tiles the HD:InsideMPW:Chapter1 and HD:InsideMPW:Chapter2 windows in a horizontal arrangement. But the command

```
TileWindows -v {Active} {Target}
```

arranges the top two windows vertically.

When the TileWindows and StackWindows commands are executed by the Tile Windows and Stack Windows menu items, the windows are tiled in accordance with the options and parameters defined in the {TileOptions} and {StackOptions} shell variables. The {TileOptions} and {StackOptions} variables are defined in the MPW Startup script. So you can change their definitions by editing the Startup script.

There are two ways to modify variables that are defined in the Startup script. You can type in new definitions directly from your keyboard, as explained in Chapter 2, or you can use the UserVariables Commando dialog, which is covered later in this chapter.

▶ The Project Menu

The Project menu, shown in Figure 3-17, provides an interface to Projector, a project-management tool built into MPW. With Projector, you can maintain a revision history of any software development project

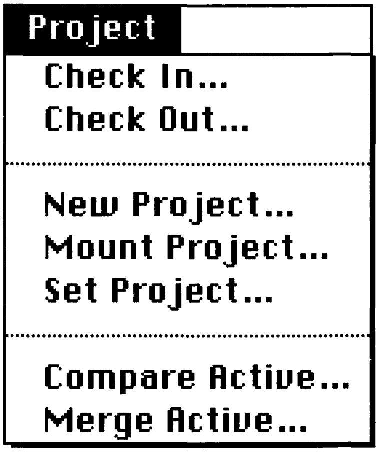


Figure 3-17. Project menu

with a backup of each revision filed away for safekeeping. You can even try out experimental versions of a program by using a branching function that keeps experimental versions of programs or program segments separated from your main development path.

Projector can be used by teams of programmers working on large projects with the aid of a file server, as well as by one programmer working on an individual project. In fact, Apple's own development engineers use Projector to write and maintain Toolbox and system software for the Macintosh.

▶ The Directory Menu

With the Directory menu, you can find out what the current (default) directory is, or change the current directory.

The Directory menu (Figure 3-18) has two items that are permanently displayed, plus a list of pathnames that change dynamically when you make a change to the default menu. The two permanent items are separated from the dynamic items by a horizontal line.

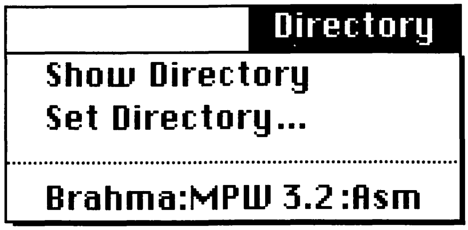


Figure 3-18. Directory menu

Show Directory

When you select the Show Directory menu item, MPW displays an alert dialog showing you the name of the current directory. The Show Directory dialog is illustrated in Figure 3-19.

Set Directory

The Set Directory menu item displays a dialog that you can use to select a new default directory. The Set Directory dialog is shown in Figure 3-20.

The Directory Menu's Pathname List

The rest of the items under the Directory menu are pathnames that have recently been selected as default directories. If you change the default directory while you are using MPW, the pathname of your new default directory is added to the list of pathnames that appear under the Directory menu. By selecting any pathname on the list, you can quickly change your current directory to the directory that you have chosen.

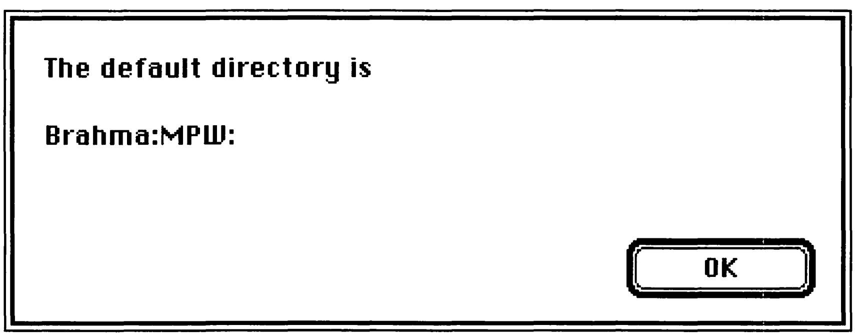


Figure 3-19. Show Directory dialog

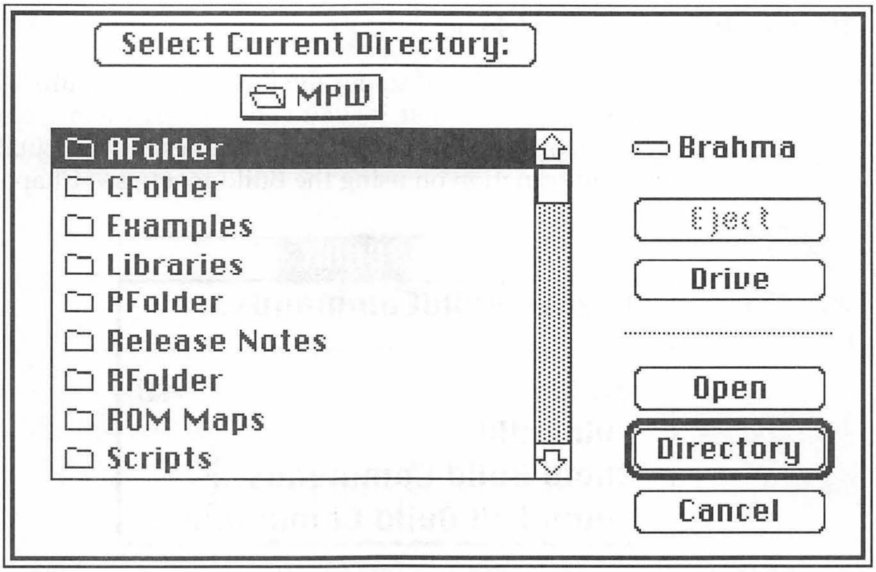


Figure 3-20. Set Directory dialog

When you launch MPW, the pathnames listed under the Directory menu are the pathnames of the folders inside the Examples folder. Figure 3-21 shows the Directory menu after MPW is launched.

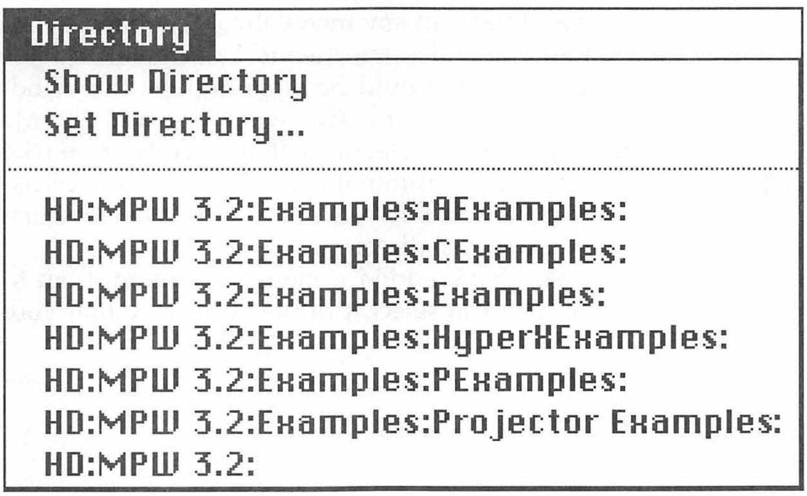


Figure 3-21. Directory menu after MPW launch

▶ The Build Menu

With the items listed under the Build menu (Figure 3-22), you can build MPW programs; that is, you can convert them from raw object code generated by a compiler or an assembler into executable programs. For more information on using the Build menu, see Chapter 8.

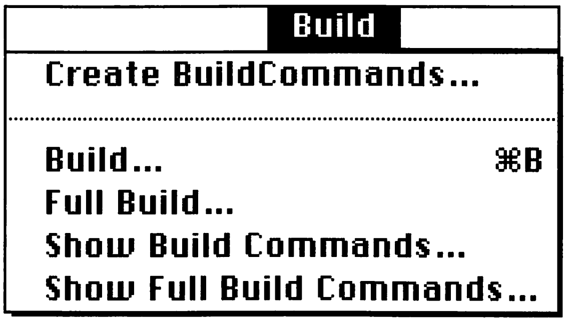


Figure 3-22. Build menu

▶ Customizing MPW Menus

You can customize the MPW menu structure by using two MPW commands: AddMenu and DeleteMenu.

With AddMenu, you can create your own menus and menu items, or you can add items to any menu already defined by MPW, except for the Mark, Window, and Apple menus. You cannot add items to those menus because doing so would be meaningless; the Window menu always contains the names of active windows, and the Mark menu always contains guides to selections that have been marked. The Directory menu also works automatically, growing longer as you shift to new default directories, but you can modify the Directory menu by adding names of additional directories.

Once you have added a menu or a menu item to the MPW menu structure, you can select it in the same way that you would select any other menu item—and it executes any command or script that you have associated with it. When you want to delete a menu or a menu item that you have defined, you can use the DeleteMenu command.

▶ The AddMenu Command

The syntax of the AddMenu command is:

```
AddMenu [menuName [itemName [command...]]]
```

where *menuName* is the name of a new or existing menu, *itemName* is the name of a new menu item, and *command* is an MPW command that you want to associate with the new menu item.

If the menu specified in the *menuName* parameter already exists, MPW adds the item *itemName* to the items listed under the existing menu. If *menuName* does not exist, MPW creates a menu with the specified name and adds it to the menu bar. The item specified in *itemName* is then added to the new menu. If both *menuName* and *itemName* already exist, the command list associated with *itemName* is changed to *command*.

▶ Omitting AddMenu Parameters

By omitting parameters, you can use AddMenu to write information about menus and menu items to the screen or to a printer. For example:

- If you do not use any parameters, AddMenu writes a list of user-defined items to standard or specified output.
- If you omit the *itemName* and *command* parameters, AddMenu writes a list of all user-defined items to standard or specified output.
- If you use the *itemName* parameter but omit the *command* parameter, AddMenu writes the command list associated with *itemName* to standard or specified output.

It is important to remember that the text that you enter in the AddMenu parameter is processed twice: once when you execute the AddMenu command itself, and again when you select the new menu item that you have added. This means that you must use the proper rules for quoting items so that they are processed at the right time. Rules for using quotation marks in MPW commands were explained in more detail in Chapter 2.

▶ Creating a Menu and a Menu Item

To illustrate the use of AddMenu, let's add a menu called Apps to your MPW menu bar so that you can launch certain applications—for example, MacPaint or MacDraw—directly from MPW by selecting their names from a pull-down menu. We'll start with MacPaint. (If you don't have MacPaint on your hard disk, you can do the exercise with another application.)

Before you can add MacPaint (or any other application) to your MPW menu tree, you must determine where it is stored on your hard disk. You could do that by looking for the application's file name on your Finder screen, but there is an easier way: You can use the MPW command WhereIs.

The WhereIs command can find any file on a disk, and writes the file's pathname to standard output. To issue a WhereIs command that finds MacPaint, execute this command line from your Worksheet window:

```
WhereIs MacPaint
```

MPW then finds the application you are looking for and prints its full pathname on your screen, like this:

```
HD:Graphics:MacPaint
```

Once you know the pathname of your MacPaint application, you are ready to create a new menu and a new menu item. To create the new menu, execute an AddMenu command in this format:

```
AddMenu Apps MacPaint "HD:Graphics:MacPaint"
```

You could accomplish the same result by substituting the {Boot} shell variable for the volume name HD, like this:

```
AddMenu Apps MacPaint '"{Boot}"Graphics:MacPaint'
```

This is a better syntax because it would still work if you changed the name of your boot volume. But when you use variables with the AddMenu command, you have to enclose them in quotation marks, as shown in this example and explained later in this chapter.

As soon as you type an AddMenu command and press Enter, the name of your new menu is added to the MPW menu bar. Pull down the Apps menu, and you should see your MacPaint item. Click on it, and MPW should launch MacPaint. You can use MacPaint for as long as you like. And, when you exit MacPaint, you're back in MPW!

Once your Apps menu has been added to the menu bar, you can add more items to it. For example, to add an item that would launch MacDraw (assuming that MacDraw is also in your Graphics folder), you could enter a command like this:

```
AddMenu Apps MacDraw 'HD:Graphics:MacDraw'
```

or

```
AddMenu Apps MacDraw "'{Boot}'Graphics:MacDraw"
```

Both MacPaint and MacDraw would then appear as items under your new menu.

▶ Adding Menus and Items from a UserStartup Script

You can execute the AddMenu command from a UserStartup script as well as from a command line. For example, if you modified your UserStartup script by adding the lines

```
AddMenu Apps MacPaint '"{Boot}"Graphics:MacPaint'
AddMenu Apps MacDraw '"{Boot}"Graphics:MacDraw'
```

your Apps menu would then be added to the menu bar every time you started MPW.

A UserStartup script that contains several AddMenu commands is presented at the end of this chapter.

▶ Using AddMenu to Run a Script

With the AddMenu command, you can create menu items that run MPW scripts as well as applications. For example, to create a menu item that would run the Chimes script (which we created in Chapter 2 when we wrote our first script) you could execute this command:

```
AddMenu Apps Chimes '"{MPW}"Chimes'
```

You could then run the Chimes script by selecting the Chimes command.

It might be helpful at this point to take note of how the quotation marks are arranged in the preceding example. They have been placed very carefully. If you typed the example this way, it would not work if {MPW} contained spaces.

```
AddMenu Apps Chimes "{MPW}Chimes"    # This might not
                                     # work
```

▶ Menu Items for Editing Documents

You can use the AddMenu command to create menu items that can perform many kinds of functions. In this section, menu items will be created to move the insertion point to the top of a document, move the insertion point to the bottom of a document, insert selected text into a document, and add selected text to a document.

By the Way ▶

Pattern-matching Characters in Menu Commands. In the AddMenu command that creates the Top menu item, the special character • (Option-8) is used as a pattern-matching character to represent the top of a document. Other special characters used for similar purposes include ∞ (Option-5), used to represent the bottom of a document; § (Option-6), used to represent the current selection); and Δ (Option-J), used to represent the beginning or end of a selection (if it appears before a selection, it represents the beginning, and if it appears after a selection, it represents the end). These and other pattern-matching characters are examined in more detail in Chapter 4.

Moving to the Top of a Document

If you add this line to `UserStartup` script

```
AddMenu Find Top 'Find • "{Active}"'
```

MPW adds to the Find menu a menu item labeled Top. Then, when Top is selected, the MPW Editor's insertion point is placed at the top of the active window. (This menu item, incidentally, has a keyboard equivalent: Command-Shift-Up Arrow.)

By the Way ▶

Doing It from the Keyboard. The Top menu item has a command-key equivalent. You can move the insertion point to the top of a document by pressing the Up-Arrow key while holding the Shift and Command keys down. To get to the bottom of a document, you can use the keyboard equivalent for the Bottom menu item, Shift-Command-Down Arrow.

Moving to the Bottom of a Document

If you find that a Top menu item is a useful addition to your Find menu, you will probably also want to add a matching item called Bottom. This Bottom command creates a Bottom menu item, which sends the insertion point to the bottom of the document in the active window:

```
AddMenu Find Bottom 'Find ∞ "{Active}"'
```

Two other custom menu items that you might find useful are Insert and Add. Insert copies selected text from the document in the active window to the insertion point in the target window. Add copies a selection from the active window and adds it to the bottom of the insertion point in the target window.

This command adds the Insert item to the Edit menu:

```
AddMenu Edit Insert ∂
'Copy § "{Active}"; ∂
Paste § "{Target}"'
```

This command creates the Add menu item:

```
AddMenu Edit Add ∂
'Find §Δ; Catenate "{Active}.§" > §'
```

▶ Using Metacharacters with the AddMenu Command

In the parameters that you pass to the AddMenu command, you can use a special set of characters called metacharacters to define the appearance and operation of any menu item. For example, the metacharacter

```
(
```

disables the item that follows it, dimming the item's name and making it unselectable. And the character

```
-
```

prints a horizontal line to separate menu items. So the command

```
AddMenu Edit "(-" " "
```

or

```
AddMenu Edit '(-' ' '
```

prints a dotted horizontal line as the next item under the Edit menu. Since no command is associated with this menu item, a space character enclosed in quotation marks is typed at the end of the command to indicate the presence of a null item.

The metacharacters that are used with the AddMenu command are the same as those that are used with the Menu Manager trap call AppendMenu. They are listed in Table 3-3.

Table 3-3. Metacharacters used with menu items

Metacharacter	*Usage*
;	Separates multiple items.
Return	Separates multiple items.
^	Followed by an icon number (included in a resource fork), adds the specified icon to the item.
!	Followed by a check mark or other character, marks the item with the specified character.

Table 3-3. Metacharacters used with menu items (continued)

Metacharacter	*Usage*
<	Followed by B, I, U, O, or S, sets the character style of the item, as follows: <B bold <I italic <U underline <O outline <S shadow
/	Followed by a character, associates a keyboard equivalent with the item.
(	Disables the item.
-	Prints a horizontal line to separate menu items.

▶ The DeleteMenu Command

The DeleteMenu command deletes any menu or menu item that has been added using AddMenu. Thus, the only preexisting menus that you can delete or modify using DeleteMenu are the Build and Project menus, which are defined in your UserStartup script.

Space on the MPW menu bar is in short supply, and it would be nice if you could grab some more space for your own menus by using the DeleteMenu command. However, if you deleted the Build menu, you would not be able to use it to build programs. If you deleted the Project menu, you would not be able to use Projector, MPW's project management tool.

If you do not want to comment out your Project menu, there is enough space at the end of the MPW menu bar for one menu name—provided you give it a short label. And, of course, you can always create as many menu items as you like if you append them to existing menus.

The syntax of the DeleteMenu command is:

```
DeleteMenu [menuName [itemName]]
```

DeleteMenu deletes the item specified in the parameter *itemName* from the menu specified in the *menuName* parameter. If no *itemName* is specified, DeleteMenu deletes all items from the specified menu. If no *menuName* is specified, DeleteMenu deletes all user-defined menus and items.

MPW Dialogs

MPW places two kinds of dialogs at your disposal: Standard dialogs and Commando dialogs. Commando dialogs derive their name from the fact that they are used to execute MPW commands. They are examined in detail later in this chapter. But first, we will look at some standard dialogs that you can create using MPW scripts and MPW commands.

Using Standard Dialogs in MPW

There are several commands that you can use to create standard dialogs quickly and easily in MPW. They include:

- Alert
- Confirm
- Request

The Alert Command

The Alert command creates an alert dialog containing a message and an "OK" button (but no alert icon). The syntax of the Alert command is

```
Alert [-s] [message...]
```

You can place any alert string you like in the *message* parameter. If you use the -s option, Alert runs silently, without emitting a beep. If you do not use -s, a beep is sounded. Alert displays a dialog like the one shown in Figure 3-23.

Consider this example of a command line containing an Alert command:

```
C "{MyFile}"Prog.c || Alert "Could not compile file"
```

If you executed this command, MPW would attempt to compile a program called Prog.c in a directory identified by the {MyFile} variable. If the compilation failed, MPW would display an alert dialog containing the message, "Could not compile file."

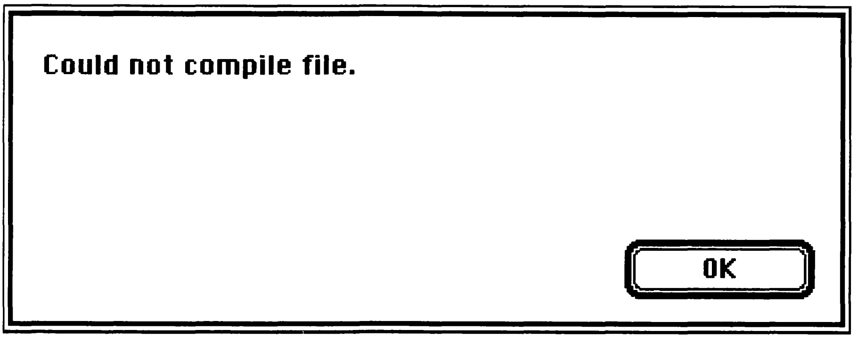


Figure 3-23. Alert dialog

The Confirm Command

Confirm displays a dialog containing a message that a user can respond to by clicking a Yes, No, or Cancel button. The syntax of the Confirm command is

```
Confirm [-t] [message...]
```

The Confirm command displays an dialog like the one illustrated in Figure 3-24. The dialog contains the message specified in the *message* parameter. If you use the -t option, Confirm displays a dialog with three response buttons: Yes, No, and Cancel. If you do not use -t, Confirm displays two response buttons: OK and Cancel.

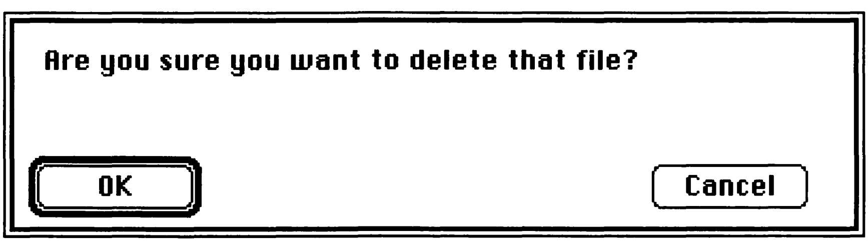


Figure 3-24. Confirm dialog

When you use Confirm in a tool or a script, it returns its result in the {Status}. shell variable. These are the values that Confirm can return:

Value	*Meaning*
0	The user clicked the OK button.
1	Syntax error.
4	The Cancel button was clicked in a two-button dialog, or the No button was clicked in a three-button dialog.
5	The Cancel button was clicked in a three-button dialog.

Important ▶

The Confirm Command and the {Exit} Variable. Since Confirm returns a nonzero status value when No or Cancel is selected, you should set the {Exit} shell variable to zero before issuing a Confirm command. This step is necessary because the shell aborts script processing when a nonzero status value is returned and {Exit} is set to nonzero.

Listing 3-1 is a short script containing a Confirm command. The script uses the Delete command to erase a file called MyFile, but asks first for confirmation.

Listing 3-1. Confirm script

```
Set Exit 0
Confirm "Are you sure you want ∂
to delete that file?"
If "{Status}" == 0
    Delete MyFile
End
Set Exit 1
```

The Request Command

The Request command displays a dialog containing a prompt and a TextEdit item. In the TextEdit box, the user can type a response to the dialog's prompt. The response is written to standard output by default, but it can be used to set a variable. A Request dialog also contains an OK button and a Cancel button.

A typical Request dialog is shown in Figure 3-25.

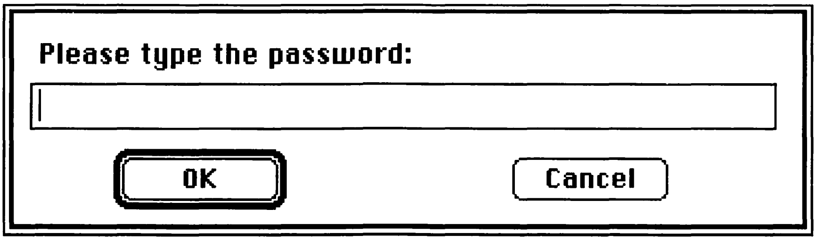


Figure 3-25. Request dialog

The syntax of the Request command is:

```
Request [-q] [-d default] [message...]
```

If the -q option is used, a Request dialog always returns a status of either zero or one. Otherwise, Request returns these values in the {Status} variable:

Value	*Meaning*
0	The OK button was selected.
1	Syntax error.
4	The Cancel button was selected.

If the -d option is used, it must be followed by a string. This string, known as the *default* parameter, appears in the response box when the dialog is displayed.

Listing 3-2, a script called Login, illustrates the use of Request command. The script displays a dialog containing the prompt, "Please type the password:". If the user responds with the correct password—the word "Karoshi"—MPW plays a melody, displays an alert that says, "Welcome to the Editor!," opens a new window, and echoes the message, "You are now online." (For the chimes to work, you must have the Chimes script—created in Chapter 2—in your MPW folder.)

Listing 3-2. Login script

```
Set Exit 0
Set N "`Request 'Please type the password:'`"
If "{Status}" == 0
    If "{N}" =~ /Karoshi/
        # This happens if login succeeds
        Chimes
        Alert -s "Welcome to the Editor!"
        New
        Echo "You are now online.∂n" > "{Active}"
    Else
        # This happens if login fails
        Alert "Password invalid; access denied."
    End
End
Set Exit 1
```

If the user fails to type the correct password, the script displays an alert that says, "Password invalid; access denied" and does not put the user online.

Although pattern-matching procedures won't be examined in detail until Chapter 4, three of the features of the Login script in Listing 3-2 are worth examining now. For example, in the line

```
If {N} =~ /Karoshi/
```

the contents of the {N} variable, set by the Request dialog, are compared to the predefined string "Karoshi." Note that in this line, the =~ operator is used to mean "is equal to." In MPW, =~ is the standard operator for comparing strings, whereas == serves the same purpose when numeric values and case-sensitive strings are being compared.

Also note that in the line the string "Karoshi" is delimited by slash bars, rather than by quotation marks. In MPW pattern-matching operations, slash bars are the standard delimiters.

A third construct in Listing 3-2 that is worth noting is .∂n, used in the line that begins with the command word "Echo". As you may recall from Chapter 2, ∂ is used as an escape character before the letters *n*, *t*, or *f*. When it appears before the letter *n*, it writes a newline character, or return, to standard output. When used before the letter *t*, it writes a tab. When used before the letter *f*, it writes a form feed. When ∂ appears by itself at the end of a line in a script, its behavior is completely different;

it then causes the line that it ends and the line that follows to be written as a single line.

In the Login script, ∂ causes a newline character to be written following the message, "You are now online." The newline moves the insertion point to the beginning of the next line so that the user can start typing on a new line.

▶ Commando Dialogs

Commando dialogs are a very special feature of MPW. With a Commando, you can execute any MPW command by clicking on items in a dialog box, instead of typing and entering the command on a command line. Figure 3-26 shows a typical Commando dialog.

MPW provides a unique Commando dialog for every one of its 120+ commands. You can invoke any Commando dialog by executing a simple one- or two-word command. So, when you want to execute a command but do not want to look up all its options and parameters, you can call up the Commando dialog that is associated with the command.

In a Commando dialog, every option and parameter of the command associated with the dialog is translated into a dialog item—a button, a check box, a text item, or a pop-up menu. To set the options and

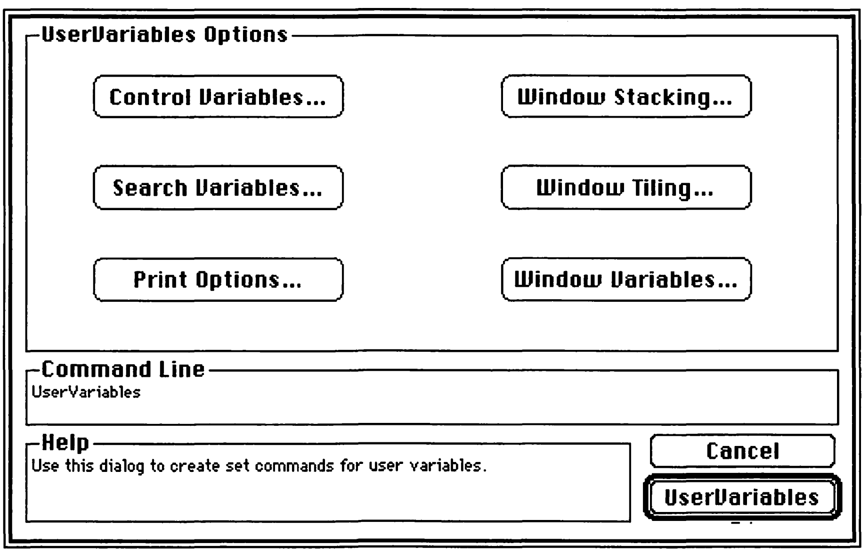


Figure 3-26. Commando dialog

parameters of the command you are executing, you can click on the options and parameters that you want to define. As you select options and parameters, the Commando dialog that you are using automatically composes a syntactically perfect command line.

When you have finished writing your command by clicking on dialog items, you can close your Commando by clicking on its OK button. Your Commando then disappears, and, depending on your preferences, either executes the command that it has composed or prints it on a line in the active window.

If you have instructed a Commando to write a command to your screen, you can execute the command in the same way that you execute a command that you have typed in. Alternatively, you can copy the command into a script for later execution.

▶ What You Can Do with Commando Dialogs

While you are learning to use MPW, Commandos are excellent educational tools. You can explore MPW commands easily and conveniently by calling up Commando dialogs and experimenting with commands and parameters by clicking on dialog controls.

Commandos can also save you a lot of work when you use MPW. When you are thinking of using a command, but are not quite sure about its syntax, its options, or its parameters, you can call up a Commando dialog. Then you can compose the command line you need by clicking the mouse in a few controls instead of looking everything up and typing it in with syntactical perfection.

▶ An Example: The UserVariable Commando

As mentioned in Chapter 2, you can set the values of MPW's user variables by executing a shell script called UserVariables. The UserVariables script invokes a Commando dialog called—what else?—the UserVariables Commando.

The UserVariables Commando can be used to set the following shell variables: {NewWindowRect}, which contains the coordinates of new windows; {ZoomWindowRect}, which defines the zoom coordinates of windows; {StackOptions}, which holds parameters for the StackWindows command; {TileOptions}, which defines parameters for the TileWindows command; and {IgnoreCmdPeriod}, which determines whether Command-Period, the standard Macintosh "Halt" keystroke sequence, is recognized during critical operations.

When you define these user variables by invoking the UserVariables Commando, you do not have to type in your variable definitions from your keyboard—and you do not have to thumb through the *MPW 3.0 Reference* to look up all the possible ways that each variable can be changed. Instead, you can let the UserVariables Commando do most of the work for you.

To execute the UserVariables script, and thus call up the UserVariables dialog, simply type the command

```
UserVariables
```

MPW then executes the UserVariables script and displays the UserVariables Commando dialog on your screen.

The dialog that was shown in Figure 3-26 is the UserVariables Commando.

▶ The Parts of a Commando Dialog

Like all Commando dialogs, the UserVariable Commando is divided into five parts:

1. An Options window
2. A Command Line window
3. A Help window
4. A Cancel button
5. An OK button containing the name of the command that was used to call the Commando dialog

The five parts of a Commando dialog are labeled in Figure 3-27.

▶ The Options Window

As you can see in Figure 3-26, the UserVariables Commando divides MPW's user variables into six categories. In the dialog's Options window, there are six buttons, one for each category.

To set the default value of a user variable, just place your cursor in the Options window and click the button that matches the category of the variable that you want to define. A second dialog then is displayed. By selecting items from that dialog, you can choose the variable that you want to define and set its options and parameters.

Suppose, for example, that you wanted to set the {TileOptions} shell variable. {TileOptions}, as mentioned earlier in this chapter and in

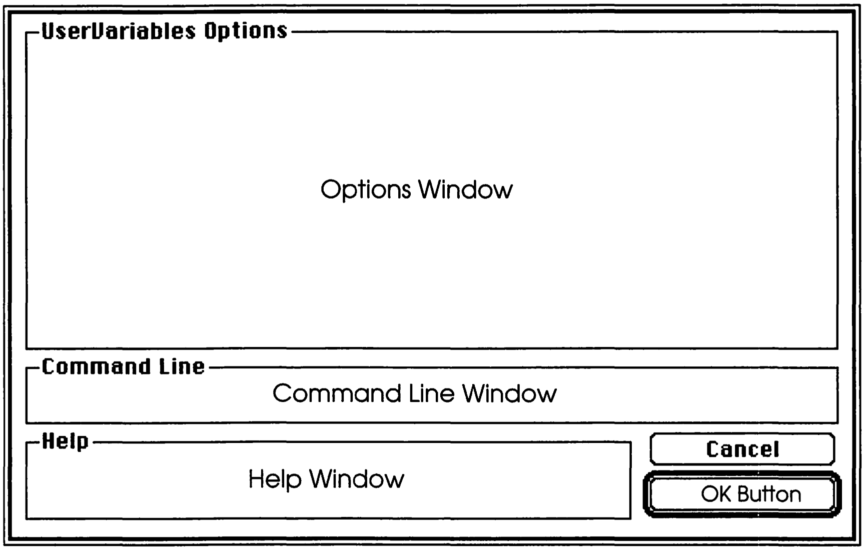


Figure 3-27. Parts of a Commando

Chapter 2, determines how windows are arranged on the screen when you tile them using the TileWindows command or the Tile Windows menu option.

To set the {TileOptions} variable with the UserVariables Commando, click on the button labeled Window Tiling in the Commando's Options window. The Commando dialog then displays a second dialog, like the one shown in Figure 3-28.

By clicking on the items in the Window Tiling dialog, you can enter information using

- check boxes to include the Worksheet window, the active window, or the target window in the tiling operation, in any combination
- radio buttons to display tiled windows in a regular (checkerboard) arrangement, a horizontal arrangement, or a vertical configuration (note the icons that the Commando provides to help you make your decision)
- a text item into which you can type the coordinates of a rectangle enclosing your tiled windows (this feature could be useful if you have a large screen)

When you have finished setting the options in the Window Tiling dialog, you can click its close box. It disappears, returning you to the UserVariables Commando. You can then click on another one of the

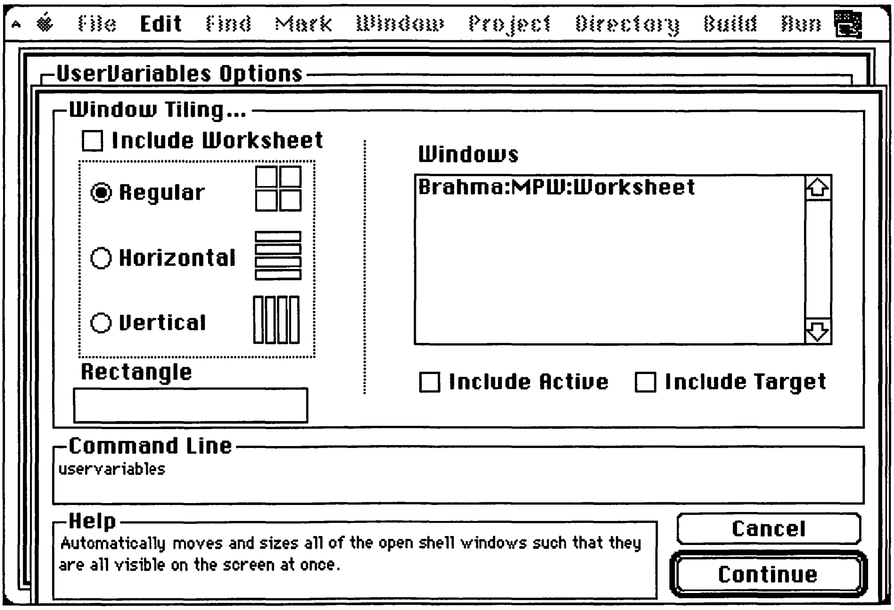


Figure 3-28. TileWindows dialog

Commando's six Options buttons or close the Commando by clicking its OK button—which, in this case, is labeled UserVariables. Alternatively, you can click on the Commando's Cancel button, aborting any action by the Commando dialog.

▶ The Command Line Window

As you click on a Commando's controls to define various options and parameters to set various variables, the Commando keeps track of what you are doing by creating a command line and displaying it in the Command Line window. Each time you add an option or a parameter by selecting another dialog item, the Commando adds your new selection to its command line.

The Commando also alters its command line when you "unselect" a dialog item. If you change your mind while you are choosing options and variables, and you undo a change that you have made, the Commando rewrites its command line to undo your previous command. Thus the command line composed by a Commando dialog always reflects the options and parameters you have selected, no matter how many times you change your mind.

▶ The Help Window

Commando dialogs have one feature that is extremely useful, but so well hidden that if you did not stumble across it, you would probably never find out about it on your own. That feature is a most unusual Help utility that is built into every Commando dialog.

To use a Commando dialog's Help utility, all you have to do is click your mouse over the *label* of a control. For example, in the TileWindows dialog shown in Figure 3-28, if you press your mouse button over the word "Horizontal"—the label of the "Horizontal" radio button—the text in the Help window changes to read, "Tile windows horizontally." Select the label of the "Vertical" button, and the text in the Help window changes to read, "Tile windows vertically."

▶ The OK Button

When you have finished writing a command using a Commando dialog, you can close the Commando and execute it by clicking on its OK button—which is always labeled with the name of the Commando's corresponding command. The Commando dialog then disappears.

When you execute the UserVariables Commando, it writes the command that it has composed to standard output, ordinarily the screen. So, if you set a variable with the UserVariables Commando and then press the Commando's OK button, you see a line similar to this printed in your active window:

```
UserVariables ; Set TileOptions " -v "
```

Notice that there are two commands on this line:

- UserVariables, which is the command you issued to invoke the UserVariables Commando
- Set TileOptions " -v ", which is the command composed by the Commando dialog

Note ▸

A Note About the UserVariables Commando. Most Commando dialogs do not print their output in this fashion; in fact, UserVariables is the only Commando that echoes your command, along with its own output, in the active window. Why UserVariables behaves in this unique fashion is explained at the end of this section.

If you want to insert a command written by the UserVariables Commando into your UserStartup script, you can copy the command that it has composed—in this case, Set TileOptions " -v "—into your UserStartup script using MPW's Edit menu or standard cut-and-paste commands. But, be careful to copy only the useful part of the Commando's output; that is, the portion of the line that comes after the semicolon. If you do not want to take the trouble to copy a command into your UserStartup script, there is no need to. There is a handy script that can simplify this operation. That script, called UserVar, is described and listed later in this chapter.

▶ The Cancel Button

If you call up a Commando and then decide you want to close it without taking any action, you can click in the Commando's Cancel button. That aborts the Commando dialog.

▶ Some Unique Features of the UserVariables Commando

Before we see how to invoke a Commando dialog, we should note that although UserVariables is a command, it is not an official shell command. Rather, it is a script that you can execute as a command by typing and entering its name.

The UserVariables script has just one function: to invoke the UserVariables Commando. The UserVariables command is included in MPW because it provides a method for defining a whole set of variables from one Commando dialog.

UserVariables is the only MPW script that was written for the sole purpose of calling a Commando dialog. The UserVariables Commando is the only MPW Commando that is used to define a set of variables. Every other Commando in MPW executes a single command.

In addition to being the only Commando that is called by a script, UserVariables is also the only Commando that can be invoked by typing and entering just one word: the name of the script that calls it. The procedures for invoking all of MPW's other Commando dialogs are described in the next section.

As pointed out in the box "A Note About the UserVariables Commando," the UserVariables dialog also writes its output to the screen in an unusual fashion: It is the only Commando that echoes your command, along with its own output, in the active window. All other Commandos echo only their own output to the screen, making it considerably easier for you to copy into scripts the commands they have composed.

As mentioned at the beginning of this section, you can ordinarily instruct a Commando dialog to *execute* the command that it writes, instead of printing it on the screen. Again, however, the UserVariables Commando is an exception. It is designed to write to standard output, normally the active window, so that its output can be pasted into the UserStartup script.

▶ Improving the UserVariables Script

But there is a way to improve the operation of the UserVariables script—with a script I wrote called UserVar. It is shown in Listing 3-3. It is a bit too complex to be analyzed line by line at this point. However, an explanation of how it works is presented at the end of Chapter 4.

Listing 3-3. UserVar script

```
# UserVar Script
#
# Writes the output of the UserVariables Commando
# To your UserStartup Script
#
# Just Execute the Command UserVariables
# And run this script;
# It will do the rest

Find ∞ "{Active}"   # Go to end of window
## Search backwards for output of UserVariables
Find \UserVariables ; \Δ "{Active}"
Find §:∞ "{Active}"     #Select it
## Open UserStartup script
Open "{ShellDirectory}"UserStartup
Find ∞ "{ShellDirectory}"UserStartup    # Go to end
## Write a return to start a new line
Echo "∂n" >> "{ShellDirectory}"UserStartup
## Add output of UserVariables to UserStartup
Catenate "{Target}".§ >> "{ShellDirectory}"UserStartup
```

To improve the UserVariables script with my UserVar script, all you have to do is open your Worksheet window and execute the two commands

```
UserVariables; UserVar
```

The first command, UserVariables, opens the UserVariable Commando dialog. Then you can use the UserVariables Commando and its various subdialogs to set as many variables as you wish.

When you close the Commando, my UserVar script takes over. It opens your UserStartup script and appends to it all the variable definitions that you have set using the UserVariables Commando.

▶ Executing UserVar from the Menu

If you like the way the UserVar script works, you can create an MPW menu item that calls it for you so you will not have to execute it from a command line. Then, when you want to add a new variable setting to your UserStartup script, all you have to do is select a menu item and define your variable using the UserVariables Commando. The rest of the work is done by the UserVariables and UserVar commands.

You can create a UserVar menu item this way:

1. Copy the UserVar script in Listing 3-3 into the MPW folder Scripts.
2. Create a menu item to run the UserVariable and UserVar scripts by inserting a line like this into your UserStartup script:

```
AddMenu Cmdo 'Set Variables' 'UserVariables ; UserVar'
```

This command is included, incidentally, in the modified UserStartup script shown in Listing 3-4.

▶ Invoking a Commando Dialog

Every Commando dialog except the UserVariables Commando can be called in three ways:

1. By typing Option-Enter or Command-Option Return
2. By typing the name of a command followed by an ellipsis (Option-;)
3. By typing the word "Commando," followed by the name of a command

The first two methods of calling a Commando achieve the same results: They *execute* the command created by the Commando that they call. The third method is different: Instead of executing the command written by a Commando, it merely writes the command to standard output, normally the screen.

When you invoke a Commando using Option-Enter or an ellipsis, you are implicitly executing the Commando command, which creates Commando dialogs. When you execute a Commando by typing the Commando command, you are explicitly executing the Commando command.

Calling a Commando with Option-Enter

When you want to invoke a Commando dialog interactively, the easiest method is to type the name of a command and then press Option-Enter, using the Enter key on your numeric keypad. For example, you can call the Commando for the SetFile command by typing

```
SetFile
```

and then pressing Option-Enter. SetFile displays a dialog like the one in Figure 3-29.

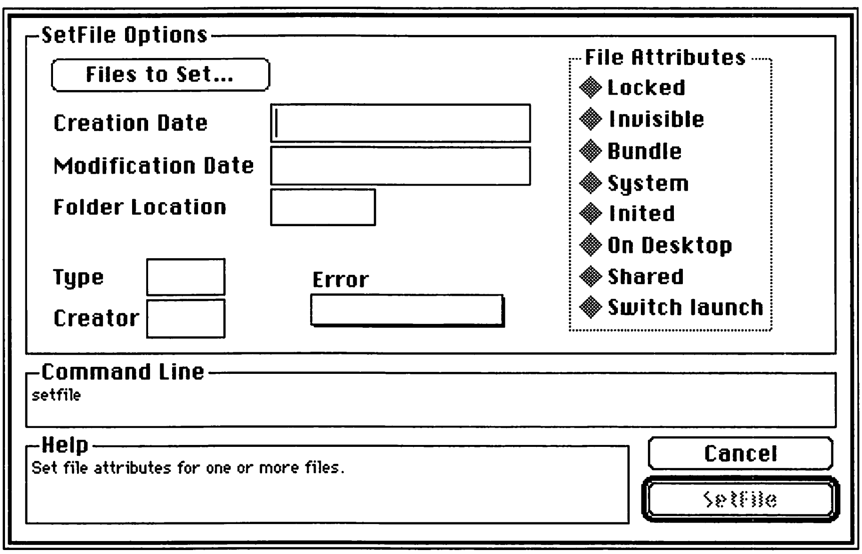


Figure 3-29. SetFile Commando

▶ Calling a Commando with Option-;

To call a Commando using the ellipsis (Option-;) character, you can use this format:

```
Command...
```

For example, to invoke the SetFile Commando, you can type:

```
SetFile...
```

and press the Enter key on your numeric keypad, *without* holding the Option key down. You can also use the ellipsis format in a script.

Important ▶

Typing an Ellipsis. When you type an ellipsis to invoke a Commando dialog, remember that an ellipsis may look like three periods, but it is not. To create an ellipsis, you cannot type three periods; you *must* type Command-semicolon.

When you invoke a Commando with an ellipsis command, the ellipsis may appear anywhere on your command line except within quotation marks or after the escape character ∂ (Option-D), and it is considered a word-break character.

Option-Enter and Option-; Recognize Aliases

When you call a Commando using either Option-Enter or an ellipsis, the Commando is not invoked until the shell has carried out all alias and variable substitutions. That means that when you invoke a Commando using Option-Enter or an ellipsis, you can use an alias to enter the name of the command that you want to execute. For example, if you have assigned the Type alias to the Echo command you could invoke the Echo Commando by entering the command

```
Type
```

followed by Option-Enter, or by executing the command

```
Type...
```

(Type followed by an ellipsis character), without holding the Option key down.

Option-Enter and Option-; Execute Commands

Also, as mentioned earlier, when you close a Commando that you have called using an Option-Enter command or an ellipsis command, the Commando *executes* the command line that it has written, rather than echoing to your screen.

Calling a Commando with the Commando Command

To invoke a Commando by explicitly calling the Commando command, use this format:

```
Commando commandName
```

where *commandName* is the name of the command you want to execute. For example, to invoke the SetFile Commando, you can execute the command

```
Commando SetFile
```

Commando Does Not Recognize Aliases

When you invoke Commando explicitly, you cannot use an alias on your command line. If you try, Commando will not be able to find your alias and thus will not display a Commando dialog.

Commando Does Not Execute Commands

Also, when you close a Commando that you have invoked by typing the Commando command, the Commando will not execute the command that it has written. Instead, it will only echo its output to your screen.

In some cases, that is exactly what you want a Commando to do. For example, you may want to copy a Commando's output to a script, but you may not want the output executed until the script is run. In a case such as this, the format

```
Commando commandName
```

is the one to use.

▶ The SetFile Commando

Another Commando dialog that is worth taking a closer look at is the SetFile Commando, shown in Figure 3-29. The SetFile Commando, unlike the UserVariables Commando, is a traditional Commando dialog that is designed to execute a single command. It sets the options and parameters of the SetFile command and either echoes the command or echoes it and then executes it, depending on whether you have implicitly or explicitly executed the Commando command.

The SetFile command was described in Chapter 2. It is used to set the attributes of files. The attributes that it defines are shown in Figure 3-29.

One noteworthy feature of the SetFile Commando is a set of three-state buttons that appears in its upper right-hand corner. Three-state buttons are diamond-shaped. They behave like check boxes, except that they have three settings instead of two. Three-state buttons, like check boxes, are used to set Boolean values. If a three-state button is black, the Boolean value with which it is associated is set to true. If the button is white, its value is set to false. If it is gray, its value is left unchanged.

▶ The Commando Commando

The last Commando that is examined in this chapter is the Commando Commando: the Commando dialog that is invoked by the Commando command. It is shown in Figure 3-30, in its full Commando gear.

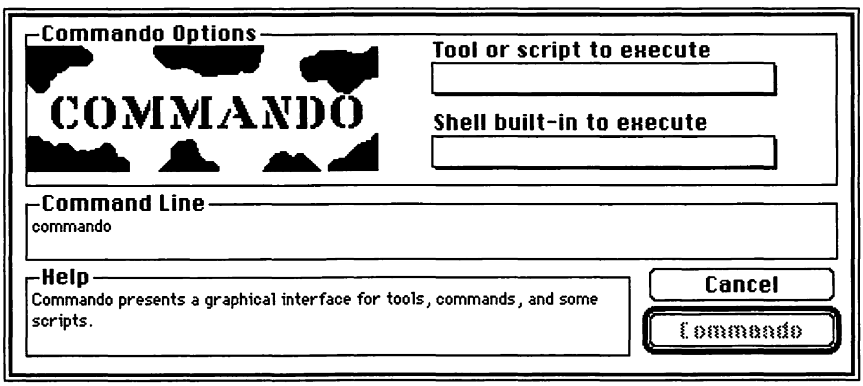


Figure 3-30. "Commando" Commando

The Commando Commando has two pop-up menus: one labeled "Tool or script to execute" and the other labeled "Shell built-in to execute."

If you select "Tool or script to execute," the Commando displays a Standard File Manager dialog, which you can use to select a user-written tool or script. If you pick a tool from the list that has a Commando dialog associated with it, the Commando Commando then invokes the appropriate Commando dialog.

If you select the pop-up menu labeled "Shell built-in to execute," you are presented with a list of shell commands. When you pick a command from the list, the Commando dialog that is associated with that command is displayed.

The Commando Commando, like the UserVariables Commando, is a bit unusual in its output; whether you invoke it explicitly or implicitly, it does not execute the output of any of the Commandos that it calls. It merely displays the Commando you specify and then writes the Commando's output to the screen. Once a command has been composed and written to the screen, of course, you can execute it by selecting it and pressing Enter, in the same way you would execute any command.

▶ Executing Commando Dialogs from the Menu

To execute a Commando dialog from the MPW menu bar, you can incorporate the command that calls the Commando into your UserStartup script. Listing 3-4, a modified UserStartup script, includes commands that call all three of the Commando dialogs examined in this chapter: UserVariables, SetFile, and Commando.

▶ Editing a Commando

MPW has a built-in Commando editor, which you can use to change the appearance of any Commando dialog. With the Commando editor, you can change the locations of the controls in any Commando, and you can also change their sizes.

You can enable the Commando editor by issuing the Commando command, in any of the three ways mentioned in the preceding sections, while holding down the Option key. Or you can execute the Commando command with the -modify option in either of these formats:

```
Commando commandName -modify
```

or

```
commandName... -modify
```

When you have invoked a Commando dialog with the editor activated, you can edit any control in the Commando by holding down the Option key while you press the mouse button inside the control. When you select a control in this way, a tiny gray rectangle appears in the lower left-hand corner of the control you have chosen. When the small rectangle appears, you can resize the control by holding down the mouse button inside the rectangle and dragging the corner of the control.

If you want to move the control, you can press the mouse button anywhere inside the control and drag it around.

When you close a Commando that you have been editing, the Commando editor displays a dialog asking you if you want to save the Commando in its new form. If you click the OK button, the edited Commando is saved.

▶ Creating Your Own Commandos

Although you can edit any Commando using the Commando editor, the real purpose of the editor is to allow you to create your own Commandos.

Commando dialogs are stored in memory as resources. You can create a Commando for any new tool or script by creating a resource fork for it and then copying the code for any preexisting Commando dialog into the command's resource file.

When you have copied a Commando into the resource fork of a tool or a script, you can use the Commando editor—along with the Rez, DeRez, and ResEdit commands—to add controls, edit strings, and change the appearance of the dialog until you have it just the way you want it. Then you can save the dialog, and you will have a customized Commando for your script or tool.

Resource forks and the Rez, DeRez, and ResEdit commands are covered in Chapter 6. Chapter 6 also provides more detailed procedures for creating new Commandos.

▶ A Modified UserStartup Script

Listing 3-4 is a UserStartup script that has been edited to include the menu changes and other programming aids described in this chapter. The Edit menu has been modified to include the items Top, Bottom, Insert, and Add, and new menus have been added to execute applications and Commando dialogs. At the end of the script, there is a melody that lets you know when MPW has finished loading, and there is a time and date stamp that records when your editing session began.

Listing 3-4. A Modified UserStartup script

```
#   Modified UserStartup Script
#   By Mark Andrews
#
#   (Original provided by MPW)
#

################ Customized Aliases #############################

Alias Type Echo              # write text to standard ouput
Alias ff WhereIs             # findfile
Alias Dir Files              # list files and directories
Alias CD SetDirectory        # change default (current) directory
Alias ChDir SetDirectory     # change default (current) directory
Alias Create New             # open new window (file)
Alias Cpy Duplicate          # copy a file
Alias Dup Duplicate          # copy a file
Alias cp Duplicate           # copy a file
Alias MD NewFolder           # create new directory
Alias MkDir NewFolder        # create new directory
Alias Cls Clear •:∞          # clear screen (target window)
Alias ar Lib                 # make library file
Alias cat Catenate           # shorter than Catenate
Alias cc 'C -mbg off'        # compile C program, MacsBug off
Alias cmp Equal              # compare files & directories
Alias diff Compare -b        # compare, ignoring minor...
                             # ...differences in white spaces
Alias df Volumes -l          # list volumes in long format
Alias expr Evaluate          # evaluate an expression
Alias grep Search            # good old grep
Alias ll Files -x tckrbm     # list files and directories...
                             # ...in a nice format
```

Listing 3-4. A Modified UserStartup script (continued)

```
Alias lr Files -m 5 -r       # list files, directories...
                             # ...and subdirectories
Alias ls Files -m 5          # list files in 5 columns
Alias man Help               # Help
Alias mv Move                # Move files/directories
Alias pr Print               # Easier to type
Alias rm Delete              # Two letters for six
Alias source Execute         # Execute script in current scope
Alias tar Backup             # Saves keystrokes
Alias tr Translate           # Saves more keystrokes
Alias wc Count               # Count lines and characters

##############################################################

############# Set options for TileWindows: ###################

#   Always include Worksheet window
#   in window-tiling operations
#
#   (Try this and see if you like it; if not, remove line)

 Set TileOptions " -i "

############# Custom additions to Find menu ##################

## Broken line to separate custom items from default items ##
    AddMenu Find '(-' ''

## 'Top' menu item ##Find §Δ; Catenate "{Active}.§" > §     #Add

    AddMenu Find Top 'Find • "{Active}"'

## 'Bottom' menu item ##
    AddMenu Find Bottom 'Find ∞ "{Active}"'

################ Custom additions to Edit menu ###################
###################(Insert and Add) ####################

AddMenu Edit '(-' ''
AddMenu Edit Insert 'Copy § "{Active}"; Paste § "{Target}"'
AddMenu Edit Add ∂
    'Find §Δ; Catenate "{Active}.§" > §'  #Catenate
```

Listing 3-4. A Modified UserStartup script (continued)

```
#################################################################

# Project menu is commented out (can be restored by uncommenting)
#
# AddMenu Project 'Check In…' 'CheckIn -w ≥≥ "{WorkSheet}"'
# AddMenu Project 'Check Out…' 'CheckOut -w ≥≥ "{WorkSheet}"'
# AddMenu Project "(-" ""
# AddMenu Project 'New Project…' 'NewProject -w ≥≥ "{WorkSheet}"'
# AddMenu Project 'Mount Project…' 'MountProject… ∑∑ "{WorkSheet}"'
# AddMenu Project 'Set Project…' ∂
#   '(project "`getListItem -r 10 ∂
#   `MountProject -pp -s -r∂` -d "∂`Project -q∂`" ∂
#    -m "Select a new current project:" -q`") ∑ dev:null'
#  AddMenu Project "(-" ""
# AddMenu Project 'Compare Active…' ∂
#   'CompareRevisions "{Active}" ∑∑ "{WorkSheet}"'
# AddMenu Project 'Merge Active…' ∂
#   'MergeBranch "{Active}" ∑∑ "{WorkSheet}"'

#################################################################

# Create Directory menu

# Default settings commented out
# DirectoryMenu `(Files -d -i ∂
#   "{MPW}"Examples:≈ || Set Status 0) ≥ Dev:Null` ∂
#   `Directory`

# New DirectoryMenu settings

DirectoryMenu `(Files -d -i ∂
    "{Boot}Inside MPW":≈ || Set Status 0) ≥ Dev:Null` ∂
    `Directory`

########################## Custom Menus #########################

# Create Build Menu

BuildMenu
```

Listing 3-4. A Modified UserStartup script (continued)

```
###### Create Apps menu (use your own apps and pathnames) ###########

AddMenu Apps Word '"{Boot}Word Processing:Word 4.0:Microsoft Word"'
AddMenu Apps MacPaint '"{Boot}Graphics:MacPaint 2.0:MacPaint"'
AddMenu Apps MacDraw '"{Boot}Graphics:MacDraw II:MacDraw II"'
AddMenu Apps Hypercard "{Boot}"HyperCard:HyperCard

#################### Create Commando Menu #########################

AddMenu Cmdo 'Set File Info' 'SetFile…'
AddMenu Cmdo 'Set Variables' 'UserVariables ; UserVar'
AddMenu Cmdo 'Commando' 'Commando…'

##################### Miscellaneous Goodies #######################

# Big Ben will tell you when MPW has loaded

Beep 2E,40 '2C,40' 2D,40 1G,80
Beep 1G,40 2D,40 2E,40 2C,80

# A date stamp begins your editing session

Echo "∂n" ; Echo "This editing session began"
Date ; Echo "∂n"
```

▶ Conclusion

This chapter explained how commands can be issued from the MPW menu bar; told how to use and customize the MPW menu structure; showed how you can use dialog boxes in MPW scripts; and provided a closeup look at Commando dialogs, which can be used to execute commands by selecting dialog items rather than typing and entering command lines.

4 ▸ The MPW Special Character Set

There are two ways to go about designing a computer language. You can construct it like a spoken language, so that it will be easy to learn and understand. Or you can design it using a more concise but less English-like model, so it will be faster, more efficient and, all too often, quite difficult to master.

When the creators of MPW sat down to develop a shell language, they could have made it a lot more user friendly. For instance, they could have used the word "TOP" instead of the • character to represent the beginning of a file, the word "BOTTOM" instead of the ∞ symbol to represent the bottom, and the word "SELECTION" instead of § to represent the currently selected, or highlighted, text. That kind of approach would have made learning the MPW shell language a much less formidable task than it has turned out to be.

There would have been tradeoffs, of course. Once you have mastered the MPW command language, it is much faster to type a command like •:∞ than it is to type something like FROM TOP TO BOTTOM SELECT ALL, which would be a possible alternative in a more English-like language. And a command interpreter can certainly parse three ASCII characters much faster than it could handle a long sequence of words in a more user-friendly language.

But that's really all quite academic. Like it or not, the MPW shell language is what we have, and if you want to use MPW, there is no alternative but to learn the MPW command language.

The MPW Special Character Set

Actually, the MPW language uses two sets of special characters. One set is made up of the punctuation marks and other special symbols that are printed on the Macintosh keyboard. The other set comes from the Macintosh extended character set: the set of characters that you get when you press a key on your keyboard while holding the Option key down. Those extended characters can be a real headache when you are trying to master MPW. Not only do you have to learn how they are used in the command language, you also have to figure out—and then memorize—where to find them, since they do not appear on the keyboard.

To make matters even more difficult, many special characters have more than one meaning in MPW; when they appear in one context, they mean one thing, and when they are used in a different context, they often mean another.

By the Way ▶

Improving MPW's Vocabulary. As you may recall from earlier chapters, the MPW shell language has more than 120 commands. But it also has about the same number of special characters and pairs of special characters that have specific meanings—and therefore are, essentially, words. So MPW has, in essence, a 240- to 250-word vocabulary.

A Notorious Character

The most notorious character with two meanings is undoubtedly the ∂ symbol (Option-d). When the ∂ character appears alone at the end of a line, MPW runs that line and the next line together, creating a single line. But when ∂ precedes the letter *n*, it acts as an escape character and generates a Return. Think about this for a moment, and you'll realize that the ∂ character has two meanings that are exact opposites. Sometimes it deletes a Return, and sometimes it creates one!

Many other special characters have more than one meaning in MPW. For example, an exclamation point means "not" in string and arithmetic operations, but it stands for a line of text in certain editing operations. The § character sometimes stands for the current selection (either a block of highlighted text or the current position of the cursor), and it sometimes stands for the name of a file.

One way to sort out the ambiguities in the special characters used by MPW would be to create a table listing every meaning of every special character used in the MPW command language, and then to examine that table and resolve each ambiguity that you find.

That is just what has been done in the long table that appears at the end of this chapter and as a pullout poster. In addition to listing the meanings and categories of all special characters used in the MPW language, the table shows how to type each character, describes the syntax in which each character is used, and provides an example showing how each character can be used in a command.

The table is arranged by character, not by category, so you can use it to look up the meaning of any special character, whether you know your special-character categories or not. However, to help you understand how MPW's special characters are used, this chapter divides them into the following fifteen categories.

- Blanks (spaces and tabs)
- Wildcard characters
- Command terminators
- The comment character (#)
- The line-continuation character (∂)
- The escape character (∂)
- Selection expressions
- Delimiters
- Regular expression operators
- File name generation operators
- Arithmetical and logical operators
- Number prefixes ($, 0x, 0b, and 0)
- Redirection operators
- Metacharacters used in menus
- Special characters used in makefiles

This chapter examines each of these categories separately. In addition, it introduces several important MPW commands that are often used in pattern-matching, editing, and printing operations.

By the Way ▶

Deja Vu. Some of the information in this chapter is very similar to information presented in Chapter 2. It's meant to be that way. Chapter 2 was organized tutorially, with information about MPW's special-character set scattered here and there. This chapter arranges MPW's special characters into fifteen distinct categories, so that you can look under the heading listed for any category and find all the special characters it contains. In addition, all of the special characters described in this chapter are listed alphabetically in the long table at the end of this chapter, and by category in the pullout poster.

Thus, you can use this chapter as a one-stop source for comprehensive information about MPW's special characters. The lack of such a source has been a serious flaw in previous books about MPW.

▶ Blank Characters

In MPW, a command is defined as a series of words and regular expressions separated by blanks and ending with a command terminator. There are only two blank characters in the MPW command language: Space and Tab. You can use either character to separate words in an MPW command.

If a block of text in a window is highlighted, or selected, the highlighted text is called the current selection. If no text is highlighted, the current location of the text cursor is referred to as the current selection. The location of the cursor is also referred to as the insertion point, or the point where any new selection will be inserted into the text.

Wildcard Characters

The ? and ≈ (Option-x) characters are wildcard characters in the MPW command language. The ? character matches any single character in a string, except for a return character, and the ≈ character matches any number of characters in a string except for a return. Thus the command

```
Find /Bar?/
```

selects any four-character word that begins with the letters "Bar," such as "Barb" or "Bark." The command

```
Find /Mar≈/
```

selects any word that begins with the letters "Mar," no matter how long it is. The words "Mar," "Mark," "Marlo," and "Marilyn" all qualify.

The two-character combination ?* means exactly the same thing as the wildcard ≈ character; it matches zero or more occurrences of any character pattern in a string. So the command in the preceding example could also be written

```
Find /Mar?*/
```

and it would accomplish exactly the same thing.

Table 4-1 lists MPW's wildcard characters.

Table 4-1. Wildcard characters

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
?	?	Wildcard	?	Matches any single character in a string.	Find /Bar?/	Select any four-character word that begins with "Bar."
?*	?* (same as ≈)	Wildcard	chars?*	Matches zero or more occurrences of any character (same as ≈).	Find /Mar?*/	Select any word that begins with "Mar."
≈	Option-x	Wildcard	≈	Matches any number of any characters in a string.	Find /Mar≈/	Select any word that begins with "Mar."

Important ▶

Watch Those Question Marks. Since the ? character has a special function in the shell language, you must be careful with strings that contain question marks. If a string containing a question mark is used as a parameter in an MPW command, you must either enclose the string in single or double quotation marks or precede the question mark with the escape character ∂. Single and double quotes and the use of the escape character ∂ are covered later in this chapter.

Command Terminators

The most commonly used command terminator is the Return. Unless a Return is preceded immediately by the line-continuation character ∂, it always ends a command.

Another command terminator that you will often see is the semicolon (;). By using a semicolon as a command terminator, you can type more than one command on a line. For example, if you put this line in a script

```
Beep ; Beep ; Beep
```

MPW treats it as though you had written it on three lines, like this:

```
Beep
Beep
Beep
```

and it sounds three beeps from the speaker built into your Macintosh.

The special-character combinations && and || are logical operators as well as command terminators. If you separate two commands with the && characters, the second command is executed only if the first command succeeds. Conversely, if you separate two commands with the || characters, the second command is executed only if the first command fails. For example, the line

```
Find /alpha/ && Echo Found!
```

searches for the string "alpha" in a file and echoes the exclamation "Found!" if the string is found. The line

```
Find /zebra/ || Echo Sorry!
```

echoes the message "Sorry!" if the Find command fails.

By using the command terminator | between two commands, you can pass, or pipe, the output of one command to the input of another. For example, the line

```
Files | Count -l
```

pipes a list of files to the Count command. Count then counts the number of files on the list and echoes the results to standard output, in this case, the screen.

Another good example of piping is a command that compiles and links a source file, and then reports on whether the operation succeeds:

```
C Sample.c && Link Sample.c.o -o ∂
Sample.Code || (Echo Failed; Beep)
```

In this example, the C command is used to assemble a source file written in C, and the Link command is used to link it.

If the compilation succeeds, the command links the object file that is generated by the C command. But, if either the assembly or the link operation fails, the command echoes the message "Failed," and beeps a warning. The C and Link commands are described in more detail in Chapter 5.

Table 4-2 lists the command terminators used in the MPW command language.

Table 4-2. Command terminators

Chr	*Type*	*Syntax*	*Meaning*	*Example*	*Translation*
&&	&&	c1 && c2	Executes c2 command if c1 command succeeds.	Find /charlie/ && Echo Found!	Search for string "charlie" and echo "Found!" if search succeeds.
;	;	c ; c	Treats commands on the same line as if they were on different lines.	Echo hello ; Echo goodbye	Output: (First line:) Hello (Second line:) Goodbye
Ret	Return	c (r)	Ends command.	Echo Hello(r)	Output: Hello
|	|	c1 | c2	Pipes output of c1 command to input of c2 command.	Files | Count -l	Files pipes a list of files to Count, which prints the list on the screen.
||	||	c1 || c2	Executes c2 command if c1 command fails.	Find /zebra/ || Echo Sorry!	Search for string "zebra" and echo "Sorry!" if search fails.

▶ The Comment Character

The # character precedes comments in MPW. When you place a comment on a command line, MPW ignores all text from the # character to the next command terminator. You can place the comment character at the beginning of the line, or anywhere thereafter. A comment placed at the end of command looks like this:

```
Echo This is a command      # but this is a comment
```

In an MPW script, you can "comment out" a line—that is, prevent it from executing—by typing the # character in front of the line. For example, if you had a script that executed two commands, but you wanted to eliminate the second command temporarily, you could comment out the second command like this:

```
Echo This command works
# Echo This one is commented out
```

Later, if you wanted to put the second command back into your script, you could remove the # character and restore the second command.

▶ The Line-Continuation Character ∂

When a command becomes too long for a line, you can end the line with the line-continuation character, ∂ (Option-d), and then continue your command on to the next line. MPW then treats both lines as if they were a single line. You can use as many ∂ characters as you like within a command.

To use the ∂ character as a line-continuation character, you must type a Return character immediately after the ∂, with no blanks or comments separating them. When the shell interprets the command, it discards both the ∂ character and the Return before it executes the command.

This is how the ∂ looks when it is used as a line-continuation character:

```
Echo This is one ∂
line, not two.
```

When this command is executed, Echo writes

```
This is one line, not two.
```

to standard output, normally the screen.

If a command line ends with the line-continuation character, the ∂ character has no effect on comments; they still end at the physical end of the line. For example, if you execute the command lines

```
Echo This is not two          # a comment ∂
lines, but only one.          # another comment
```

MPW writes the line

```
This is not two lines, but only one.
```

on the screen.

▶ The Escape Character ∂

The ∂ character is also used as an escape character to insert certain nonprinting characters into text. When ∂ is followed immediately by the letter *n*, it inserts a newline character, or a Return, into a document. When it is followed by the letter *t*, it inserts a Tab. When it is followed by the letter *f*, it inserts a form feed. For example, the command

```
Echo ∂n
```

prints a newline character on the screen, just as if a Return had been typed by a person typing text.

You can also prevent MPW from interpreting a special character by preceding that character with the ∂ character. For example, if you try to execute the command

```
Echo *  #This doesn't work
```

MPW responds with this error message:

```
### MPW Shell - File name pattern "*" is incorrect.
```

But, if you issue the command

```
Echo ∂*
```

MPW prints

```
*
```

to your screen.

If you enclose the ∂ character itself in quotation marks, the shell does not recognize it as a special character, but treats it like any other typed character. Table 4-3 shows the uses of the escape character ∂.

Table 4-3. The escape character ∂

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
∂	Option-d	Escape	∂n	Return	Echo ∂n	Echo a return.
∂	Option-d	Escape	∂t	Tab	Echo ∂t	Echo a tab.
∂	Option-d	Escape	∂f	Form feed	Echo ∂f	Echo a form feed.
∂	Option-d	Escape	∂s	Defeats the meaning of special character (s) that follows it.	Echo ∂¬	Output: ¬

▶ Selection Expressions

Selection expressions are special characters that can be used to find specific locations in files. Selection expressions used in MPW include • (Option-8), which represents the beginning of a file; ∞ (Option-5), which represents the end of a file; and § (Option-6), which represents the current selection. Table 4-4 lists the selection expressions used in the MPW command language.

Table 4-4. Selection expressions

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
!	!	Selection	!n	Selects the line that is *n* lines after end of current selection.	Find !3	Select the third line after the current selection.
!	!	Selection	r!n	Places insertion point *n* characters after end of regular expression.	Find /alpha/!3	Place insertion point three characters after end of the string "alpha."
¡	Option-!	Selection	¡n	Selects the line that is *n* lines before beginning of current selection.	Find ¡3	Select the third line above the current selection.
¡	Option-!	Selection	r¡n	Places insertion point *n* characters before beginning of regular expression.	Find /zebra/¡3	Place insertion point three characters before beginning of the word "zebra."
∞	Option-5	Selection	∞	End of file.	Find ∞	Place insertion point after last character in file.
§	Option-6	Selection	§	Current selection.	Print §	Print the current selection.

Table 4-4. Selection expressions (continued)

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
•	Option-8	Selection	•	Beginning of file.	Find •	Place insertion point before first character in file.
Δ	Option-j	Selection	Δr	Places insertion point before first character in regular expression.	Find Δ/charlie/	Place insertion point before first character in the word "charlie."
Δ	Option-j	Selection	rΔ	Places insertion point after last character of regular expression.	Find /charlie/Δ	Place insertion point after last character of the word "charlie."

The Characters •, ∞, and §

When the • character is used by itself, it stands for the beginning of a file. Thus the command

```
Find •
```

places the text cursor before the first character in the file in the target window. Similarly, the command

```
Find ∞
```

places the text cursor after the last character in the file in the target window.

Note ▶

Double Meanings Dept. The •, ∞, and § characters are selection expressions only when they are used alone in commands. When they are parts of expressions—that is, when they are enclosed in delimiters—they are considered regular expression operators and have different meanings. For more information, see the discussion of regular expression operators, later in this chapter.

When the § character is used alone in a command, it stands for the current selection. So the command

```
Print §
```

prints the current selection.

The Selection Character Δ

Another selection expression, Δ (Option-j), can stand for either the beginning or the end of a selection, depending on where it is placed in relation to the selection. The combination Δ*r* means, "Place the insertion point before the first character in regular expression *r*," whereas the combination *r*Δ means, "Place the insertion point after the last character of regular expression *r*." So, if you place a document like the one in Listing 4-1 in the target window, the commands

```
Find • ; Find Δ/bravo/
```

place the insertion point before the first character in the string "bravo." Similarly the commands

```
Find • ; Find /charlie/Δ
```

place the insertion point after the last character in the string "charlie."

Note that the selection expressions § and Δ can be used together. For example, the command

```
Find Δ§
```

places the insertion point at the beginning of the current selection.

Listing 4-1. A sample document

```
START
alpha
bravo
charlie
delta
echo
STOP
```

Important ▶

Using Line Numbers as Selection Expressions. You can use a line number as a selection expression in a Find command. To select a line in a file, simply use the number of the line as the parameter of a Find command. For instance, the command

```
Find 32
```

selects the thirty-second line in a file.

The ! and ¡ Characters

The selection expressions ! and ¡ (generated by pressing Option-!) are used with numbers to find specific characters in expressions or to find selections that are on specific lines. For example, the command

```
Find !3
```

selects the third line of text following the current selection. And the command

```
Find ¡3
```

selects the third line of text above the current selection. Thus, if you had a document like the one in Listing 4-1 in your target window, the commands

```
Find • ; Find !3
```

would select the line "charlie." The commands

```
Find ∞ ; Find ¡3
```

would do the same thing.

You can also use the ! and ¡ characters to place the insertion point a specified number of characters from the beginning or the end of an expression. The selection expression *r*!*n* places the insertion point *n* characters after the end of the regular expression *r*, and the selection expression *r*¡*n* places the insertion point *n* characters before the start of regular expression *r*. Thus, in Listing 4-1, the commands

```
Find • ; Find /alpha/!3
```

place the insertion point three characters after the string "alpha," or after the "r" in "bravo" (because the Return at the end of "bravo" counts as a character). The commands

```
Find • ; Find /delta/¡3
```

place the insertion point three characters before the beginning of the string "delta," or after the *l* in "charlie" (again, the Return at the end of "delta" counts as a character).

▶ Delimiters

Many kinds of delimiters are used in the MPW command language. When you want to include a space or a special character in a string or an expression, you must place the command between delimiters. The delimiters used in the MPW shell language are as follows.

- curly brackets ({...})
- single and double quotation marks
- slash bars (/.../) and backslashes (\...\)
- square brackets ([...])
- European quotes (« and »)
- parentheses

Delimiters used in the MPW command language are listed in Table 4-5.

Table 4-5. MPW Delimiters

Chr	*Press*	*Category*	*Usage*	*Meaning*	*Example*	*Translation*
"	"	Delimiter	"s"	Delimits a string in which each character is taken literally, except for ∂, {}, and \`.	Echo "{MPW}" >> "{Target}"	Echo the contents of the shell variable {MPW} to the target window.
'	'	Delimiter	's'	Delimits a string in which all characters are taken literally.	Echo '{MPW}' >> "{Target}"	Echo the string "{MPW}" to the target window.
(	(	Delimiter	(p)	Delimits a group of characters that form a pattern.	Find /("*")+/	Select a group of one or more asterisks.
)	)	Delimiter	(p)	Delimits a group of characters that form a pattern.	Find /("*")+/	Select a group of one or more asterisks.
/	/	Delimiter	/r/	Searches forward and selects regular expression.	Find /delta/	Search forward and select the string "delta."
»	Option-Shift-\	Delimiter	«n»	Delimits number standing for number of occurrences.	Find /[∂t]«2»/	Select exactly two tabs.
»	Option-Shift-\	Delimiter	«n,»	Delimits number standing for at least *n* occurrences.	Find /[∂t] «2,»/	Select two or more tabs.

Table 4-5. MPW Delimiters (continued)

Chr	*Press*	*Category*	*Usage*	*Meaning*	*Example*	*Translation*
»	Option-Shift-\	Delimiter	«n1,n2»	Delimits number standing for *n* to *n* occurrences.	Find /[∂t] «2,4»/	Select two to four tabs.
«	Option-\	Delimiter	«n»	Delimits number standing for number of occurrences.	Find /[∂t] «2»/	Select exactly two tabs.
«	Option-\	Delimiter	«n,»	Delimits number standing for at least *n* occurrences.	Find /[∂t] «2,»/	Select two or more tabs.
«	Option-\	Delimiter	«n1,n2»	Delimits number standing for *n* to *n* occurrences.	Find /[∂t] «2,4»/	Select two to four tabs.
[	[	Delimiter	[...]	Delimits a pattern.	Find /[A-F]/	Search for any character in the set A-F.
\	\	Delimiter	\r\	Search backwards and select regular expression.	Find \alpha\	Search backwards and select the string "alpha."
]	]	Delimiter	[...]	Delimits a pattern.	Find /[A-F]/	Search for any character in the set A-F.
`	`	Delimiter	c1 `c2`	Send output of c2 command to c1 command for processing.	Echo `Files - t TEXT`	Files command sends its output to Echo command, which prints the output on the screen.
{	{	Delimiter	{v}	Delimits variable v.	Echo "{MPW}"	Echo contents of shell variable {MPW}.
}	}	Delimiter	{v}	Delimits variable v.	Echo "{MPW}"	Echo contents of shell variable {MPW}.

Curly Brackets ({...})

In the MPW command language, variables are delimited by curly brackets. The only time you do not have to enclose a variable in curly brackets is when you define it using the Set command. After a variable is defined, it must always be delimited by curly brackets when it is used in a script or a command.

Single and Double Quotes

If a parameter in an MPW command contains more than one word, you must enclose the parameter in quotation marks so that MPW recognizes it as a single parameter.

If a parameter consists of only one word, quotes may be used, but they are not necessary unless the word contains a special character or a variable that contains blanks or special characters.

Differences Between Single and Double Quotes

You can use either single or double quotation marks to delimit a parameter, but MPW treats single quotes and double quotes differently. If you use single quotes around an expression, the MPW command interpreter treats every special character in the expression literally. But, if you enclose an expression in double quotes, there are three kinds of characters that are not treated literally: curly brackets ({...}), the back-quote character (`), and the ∂ character. Instead, they are interpreted as special characters.

Let's look at how single and double quotes work. During the variable expansion stage of command interpretation, the shell replaces all the undelimited variables that it finds in a command with their actual values. (Variables, remember, are enclosed in curly brackets.) The shell also expands any variables that are delimited by slash bars (/.../), backslashes (\...\), or *double* quotation marks ("..."). However, if a variable is enclosed in *single* quotation marks ('...'), like this:

```
'{MPW}'
```

it is not expanded because curly brackets are not interpreted as special characters when they are enclosed in single quotes.

The {MPW} variable, as explained in Chapter 2, is a shell variable that equates to the current pathname of the MPW folder. So, since variables delimited by double quotation marks are translated into their actual values during the variable expansion process, the command

```
Echo "{MPW}" >> "{Target}"
```

echoes the *contents* of the {MPW} variable to the target window, in this fashion:

```
HD:MPW:
```

But, because variables enclosed in single quotation marks are not expanded, the command

```
Echo '{MPW}' >> "{Target}"
```

echoes the string

```
{MPW}
```

which is a completely different result.

Errors in Using Quotes

If you want to use an apostrophe in a string and do not want it to be interpreted as a single quote, you can put double quotes around the word containing the apostrophe. Or you can precede the apostrophe with the escape character ∂.

When you use quotation marks in a command, MPW expects them to be used in pairs. Hence, if you try to execute a command that contains only one quotation character, like this,

```
Echo "One good quote deserves another    # This is
                                         # incorrect
```

MPW returns this error message:

```
### MPW Shell - "s must occur in pairs.
```

But, if you enclose your command in quotes, like this,

```
Echo "One good quote deserves another"
```

everything works out.

Nesting Quotation Marks

If a multiple-word parameter contains an apostrophe—which MPW interprets as a single quotation mark—you can prevent the parameter from generating an error by enclosing the parameter in double quotation marks, like this:

```
Echo "What's happening"
```

In response to this command, MPW echoes

```
What's happening
```

to the screen.

Conversely, a parameter that includes double quotation marks can be enclosed in single quotes, as long as it does not contain any variables, aliases, or ∂ characters. For example, the command

```
Echo 'The name of this file is "Source.c."'
```

writes

```
The name of this file is "Source.c."
```

If you want to use both single and double quotation marks in a parameter, you can use both kinds of quotes in a nested fashion. For example, the command

```
Echo '"It '"won't"' work," he declared.'
```

echoes this message:

```
"It won't work," he declared.
```

You can also place an apostrophe (a single quote) inside a pair of double quotation marks by preceding it with the escape character ∂ (Option-d), in this fashion:

```
Echo '"I don∂'t care."'
```

This command echoes the message

```
"I don't care."
```

Square Brackets ([...])

In the MPW command language, square brackets are used to delimit patterns. For example, the command

```
Find /[A-Z]/
```

finds and selects any character from *A* through *Z*. If the shell variable {CaseSensitive} is set to 0, or false—which is its default—the command

```
Find /[A-Z]/
```

also selects any lowercase character from *a* through *z*. However, if you have set {CaseSensitive} to 1, or true, you must execute a command such as

```
Find /[A-Za-z]/
```

to include both uppercase and lowercase letters in your selection criteria.

To select any uppercase or lowercase letter, or any digit from 0 through 9, when {CaseSensitive} is set to true, you can execute a command such as

```
Find /[A-Za-z0-9]/
```

The order in which you arrange your selection criteria is not significant because square brackets are used to select patterns, not specific sequences of characters. To select a specific string of characters, you can simply enclose it in quotation marks—or, in more complex cases, place it inside parentheses, as explained later in this chapter.

European Quotes (« and »)

The European quotation marks « and » (Option-\ and Option-Shift-\) are used to enclose numbers specifying the number of times that an operation is repeated. There are three ways to use European quotes in MPW commands:

1. «*n*» repeats an operation *n* times
2. «*n*,» repeats an operation at least *n* times
3. «*n*1,*n*2» repeats an operation at least *n*1 times, and at most *n*2 times

For example, the command

```
Find /[∂t]«2»/
```

finds and selects exactly two Tabs, whereas the command

```
Find /[∂t]«2,»/
```

selects any sequence of two or more Tabs. The command

```
Find /[∂t]«2,4»/
```

finds and selects any sequence of two to four Tabs.

Parentheses

Parentheses are often used to separate a string or an expression from other elements in a command, so that it is treated as a single unit and does not get mixed up with other strings or expressions.

Strings and expressions enclosed in parentheses are often used with the regular expression operators * and +, which are described later in this chapter.

When * is used as a regular expression operator, it means "zero or more." When + is used as a regular expression operator, it means "one or more." Thus the command

```
Find /("*")+/
```

selects a group of one or more asterisks. A more complex command

```
Find /[A-Za-z]+(, [∂t∂n ]* [A-Za-z]+)*/
```

selects a word that is made up of one or more alphabetic characters and is separated from other words by blanks and optional commas. Commas and blanks are not selected.

The Backquote Delimiter

The backquote character (`) is a special-purpose delimiter used in command substitution. By placing a command between a pair of backquote characters, you can pass its output to another command. For example, when you execute the command

```
Echo `Files -t TEXT`
```

the Files command compiles a list of files of the type TEXT and passes the list to the Echo command, which then prints its output on the screen.

The backquote delimiter is often used in statements containing the Evaluate command. For instance, the commands

```
Set a 64978
Set b 24935
Echo `Evaluate {a} + {b}`
```

echo the result

```
89913
```

The / and \ Delimiters

The slash bar (/) and the backslash (\) are often used as delimiters with the Find, Search, and Replace commands. When a string or expression delimited by slash bars (/.../) follows a Find command, Find searches in a forward direction. But, when Find is followed by a string or expression enclosed in backslashes (\...\), the search goes backwards. Hence, to start at the beginning of a file and search for the beginning of the string "charlie," you can execute the commands

```
Find • ; Find Δ/charlie/
```

But, if you want to start at the end of a file and search backwards for end of the string "charlie," you can execute the commands

```
Find ∞ ; Find \charlie\Δ
```

The Replace Command

The Replace command replaces the current selection with a specified string, pattern, or expression. Its syntax is:

Replace [-c *count*] *selection replacement* [*window*]

Replace searches for the *selection* parameter in the specified window and, if it is found, replaces it with the *replacement* parameter. If no window is specified, the Replace operation takes place in the target window. If a *count* parameter is specified, the operation is performed *count* times.

Status codes returned by the Replace command are:

Status Code	*Meaning*
0	At least one instance of the selection was found.
1	Syntax error.
2	Any other error.

When the Replace command appears in a command or a script, its selection parameter must be delimited by slash bars or backslashes. If the command's *replacement* parameter contains any blanks, it must be delimited by double or single quotation marks.

For example, the command

```
Replace -c ∞ /charlie/ "charlie brown"
```

replaces every occurrence of the string "charlie" with the string "charlie brown." Since the command has no *window* parameter, the operation takes place in the target window.

This command

```
Replace -c ∞ /•[ ∂t]+/ '' Test
```

strips away any blanks that may begin lines in the Test window and replaces each series of blanks with the null string. This operation removes all spaces and tabs from the beginnings of lines.

Note ▶

From Here to ∞. When the ∞ (Option-5) character follows an option that calls for a numerical value, it is a regular expression operator that stands for an infinite number. Thus the command

```
Replace -c ∞ /charlie/ "charlie brown"
```

means, "Replace all occurrences of the string 'charlie' with the string 'charlie brown'." The use of ∞ as a regular expression operator is covered later in this chapter.

The Search Command

The Search command works much like the Find command; but it searches files for patterns, rather than searching through text in open windows. The syntax of the Search command is:

```
Search [-s | -i] [-r] [-q] [-f file] pattern [file...]
```

The *file* parameter is a file name or a series of file names. The Search command searches the input files for lines that contain a pattern and writes those lines to standard output. If no file is given, standard input is searched. The *pattern* parameter is a regular expression. It must be enclosed in forward slashes (/) if there are blank characters in the expression. If the pattern contains no blank characters, the slash delimiters are optional.

When the Search command discovers a match, the name of the matching file and the line number of the matching line are echoed at the beginning of each line of output.

Options accepted by the Search command are:

Option	*Meaning*
-b	Break "File/Line" from matched pattern (MPW 3.2)
-i	Case-insensitive search (overrides {CaseSensitive})
-s	Case-sensitive search (overrides {CaseSensitive})
-nf	Write "pattern not found" to standard error and set status = 2 (MPW 3.2)
-r	Write non-matching line to standard output
-q	Suppress file name and line number in output
-f file	Lines not written to output are put in this file

Status codes returned by the Search command are:

Status Code	*Meaning*
0	No error
1	Syntax error
2	Pattern not found

This simple Search command

```
Search /gumbo/ MyFile.c
```

searches the file MyFile.c for the pattern "gumbo." All lines containing this pattern are written to standard output.

This more complex example

```
Search -f NoMatchFile /charlie brown/ SourceFile
```

writes all lines in SourceFile that contain the pattern "charlie brown" to standard output. All other lines are placed in a file named NoMatchFile. This operation separates matches from nonmatches and places them in separate files.

The following example

```
Search -q PROCEDURE ≈.p
```

uses the -q ("quiet") parameter to suppress the filenames and line numbers in the command's output, producing a single output file containing matching lines. This command uses the wildcard character ≈ to specify all files ending with the suffix ".p." Wildcard characters are examined later in this chapter.

Consider this final example

```
Search /Alias/ "{MPW}"Startup "{MPW}"UserStartup
```

That lists the Alias commands in the StartUp and UserStartup files.

▶ Regular Expression Operators

In the MPW command language, a regular expression is a combination of text characters and special characters that equates to text. MPW is equipped with a large set of regular expression operators that perform various operations on regular expressions. Table 4-6 lists the regular expression operators used in the MPW shell language.

Table 4-6. Regular expression operators

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
!~	!~	Regular expression operator	"s1" !~ /s2/	True if s1 is not equal to s2.	Print \`Evaluate "alpha" !~ /beta/\`	Output: 1
*	*	Regular expression operator	r*	Matches zero or more occurrences of regular expression.	Find /('*')+ '/'[∂r∂t]*/	Select a group of one or more asterisks followed by a slash bar and 0 or more white spaces.
+	+	Regular expression operator	r+	Matches one or more occurrences of regular expression.	Find /('*')+/	Select a group of one or more asterisks.
-	-	Regular expression operator	c1-c2	Stands for range of characters between c1 and c2.	Find /[A-Za-z] +∂n/	Select any word made up of upper- and lowercase letters that appears at the end of a line.
=~	=~	Regular expression operator	"s1" = ~ /s2/	True if s1 equals s2.	Evaluate "beta" =~ /beta/	Output: 1
:	Colon	Regular expression operator	s:s	All text between (two selections).	Find •:∞	Select (highlight) all text in file.
∞	Option-5	Regular expression operator	cmd -c ∞	(With command that takes a -c option): Repeats command to end of file.	Replace -c ∞ /123/ 456	Replace string "123" with string "456" every time it appears in target window.
∞	Option-5	Regular expression operator	r∞	Selects regular expression at the end of a line.	Find /arlie∞/	Select the letters "arlie" at the end of a line.
•	Option-8	Regular expression operator	•r	Selects regular expression at the beginning of a line.	Find /•ch/	Select the letters "ch" at the beginning of a line.
…	Option-;	Regular expression operator	c…	Executes Commando command, invokes Commando dialog for command c.	TileWindows…	Invoke TileWindows Commando.

Table 4-6. Regular expression operators (continued)

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
¬	Option-l	Regular expression operator	[¬list]	Any character not in the list.	Replace -c ∞ /[¬A-Za-z∂n" "] / "*"	Replace all characters except A–Z, a–z, Returns, and spaces with asterisks.
®	Option-r	Regular expression operator	r®n	Tags regular expression with a number (range: 1–9).	Replace / ([a-zA-Z]+)®1 [ ]+([a-zA-Z]+) ®2/ '®2 ®1'	Reverse the order of two words separated by one or more spaces.

The * and + Operators

When the * character is used in a regular expression, it means "zero or more occurrences of." When the + character is used in a regular expression, it means "one or more occurrences of." For example, the command

```
Find /('*')+'/'/
```

selects a group of one or more asterisks followed by a slash, and the command

```
Find /('*')+'/'[∂r∂t ]*/
```

selects a group of one or more asterisks followed by a slash bar and zero or more blank characters.

The =~ and !~ Operators

In matching regular expressions, the =~ operator is true if two strings or expressions match, and the !~ operator is true if two strings or expressions do not match. (In logical operations involving numbers, the == and != operators are used for equivalent purposes. Logical operations using the == and != operators are covered later in this chapter.)

When the =~ operator is used in a regular expression, its syntax is

```
"s1" =~ /s2/
```

and the statement in which it is used is true if string *s*1 is equal to string *s*2. For example, the command

```
Evaluate "beta" =~ /beta/
```

echoes 1, or true, because the operands of =~ match.

A more useful example is the sequence

```
Set alpha a
Set beta b
Evaluate "{alpha}" =~ /{beta}/
```

in which "alpha" and "beta" are variables rather than constants. In this case, the command echoes the output

```
0
```

which is MPW's value for false, because the strings "alpha" and "beta" are not the same.

To compare a variable with a constant, you could use this pair of commands:

```
Set w "alpha"
Evaluate "{w}" =~ /"alpha"/
```

The result of this operation is 1, or true, since the evaluated strings match.

Several features of this last example are worth pointing out. First, note that in the second line, the {w} variable in enclosed in quotation marks. That means that if the string equating to the variable contained blanks, the command would still work. Note also that slash bars—not quotation marks—are used to delimit the Evaluate command's second parameter. This is an unusual use for the /.../ delimiters, which are seen more often enclosing the parameters of the Find, Replace, and Search commands.

The ∞ Character

You may recall that the special character ∞ (Option-5) stands for the beginning of a file when it is used as a selection expression. It can also be used as a regular expression operator—and in regular expressions, it has two different meanings, depending on its context. It seems that the infinity character can cause an infinity of confusion. It has no less than three different meanings in the MPW command language!

When the ∞ character is used with a command operation that calls for a number—for example, in a command such as

```
Replace -c ∞ /123/ 456
```

it stands for an unlimited, or infinite, number. Thus the Replace command replaces every occurrence of the string "123" with the string "456."

When the ∞ character follows a regular expression, its meaning is quite different; then it stands for the end of a line. For example, the command

```
Find /arlie∞/
```

selects the string "arlie" at the end of a line.

The • Operator

The • character (Option-8), which stands for the beginning of a file when it is used as a selection expression, also has another meaning when it is used as a regular expression operator. But its confusion quotient is not as high as that of the character ∞.

The • character has only one meaning when it is used as a regular expression operator; when it precedes a regular expression, it stands for the beginning of a line.

For example, the command

```
Find /•char/
```

selects the string "char" at the beginning of a line.

The ¬ Character

The ¬ character (Option-l) is an interesting regular expression operator. It stands for any character that is *not* in a list. For example, the command

```
Replace -c ∞ /[¬A-Za-z∂n" "]/ "*"
```

replaces all characters except A–Z, a–z, returns, and spaces with asterisks.

The Tag Operator ®

When you use one or more strings or expressions in an MPW command, you can assign each one a reference number by following it with the tag operator ® (Option-R). Once you have assigned a tag number to a string or expression, you can refer to it by its number later in a script. MPW then recognizes the tagged string or expression by its number and can perform any operations on it that you want performed.

To use the ® operator, you must place it after a string or expression and follow it with a number. You can then use the tagged string or expression in many different kinds of operations. For example, the *selection* parameter in the command

```
Replace -c ∞ /(∂$[A-F0-9]+)®1/ '(®1)'
```

tags the pattern ∂$[A-F0-9]+ (any hexadecimal number) with the tag number ®1. Then the *replacement* parameter places parentheses around each occurrence of the tagged pattern.

If you performed such a tagging operation on a document that looked like this:

```
MemDoc     $945B
ViewList   $3B6D
HexBase    $2E4C
NumTab     $95B0
```

you would wind up with a document like this:

```
MemDoc     ($945B)
ViewList   ($3B6D)
HexBase    ($2E4C)
NumTab     ($95B0)
```

You can also use the ® operator to reverse the order of two columns in a list. For example, if you had a list like this:

```
MemDoc     ($945B)
ViewList   ($3B6D)
HexBase    ($2E4C)
NumTab     ($95B0)
```

the commands

```
Find • ; Replace -c ∞ /(≈)®1∂t(≈)®2/ ®2∂t®1
```

would convert it to a list like this:

```
($945B)   MemDoc
($3B6D)   ViewList
($2E4C)   HexBase
($95B0)   NumTab
```

The ® operator can be very useful when you want to convert a document from one format to another. For example, suppose you had a document that had been produced on a word processor, and you did not have a newline character at the end of each line of text. If you tried to place such a document in an MPW window, it would run off the right-hand edge of the screen. You would not be able to read it because MPW does not wrap words at the ends of lines.

You could use the ® operator to convert such a document into a format that would work with MPW. For example, this command

```
Replace -c ∞ /(?«60,70»)®1 [ ∂t]/ "®1∂n"
```

looks for the first space or tab that falls between the sixtieth and seventieth character on a line. At that point, it converts the line into a line that ends with a newline character.

The Ellipsis Operator (…)

The ellipsis character (Option-;) is a regular expression that displays a Commando dialog and executes the dialog's output as a command. Although the ellipsis character looks like three periods (…), it is actually one character that must be generated by pressing Option-;. The ellipsis character is described in detail in the Commando dialog section of Chapter 3.

▶ File Name Generation Operators

In the MPW shell language, there are eight special characters that can be used to perform special functions. These characters are known as file name generation operators. They are:

```
?  ≈  [ ]  *  +  « »
```

These eight characters can also be used as operators in regular expressions. If they appear in expressions that are delimited by single or double quotes, or by the / or \ slash delimiters, MPW interprets them as regular expression operators. If they are not quoted, MPW interprets them as file name generation operators.

File name generation operators have the same meanings in MPW file names that they have when they are used as regular expression operators in quoted expressions. The ? and ≈ characters are wildcard characters; the [and] characters are brackets that enclose patterns; the * and + characters stand for repetitions of a specified character; and the « and » characters enclose numbers that specify the number of times an operation is to be performed.

The file name generation operators used in the MPW command language are listed in Table 4-7.

Table 4-7. File name generation operators

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
*	*	File name operator	n*	Matches zero or more occurrences of the preceding character or character list.	X*	Match zero or more occurrences of the character X.
+	+	File name operator	r+	Matches one or more occurrences of the preceding character or characters.	X+	Match one or more occurrences of the character X.
?*	?* (same as ≈)	File name operator	?*	Matches any number of any characters in a file name.	?*.c	Any file name with the extension ".c".
¬	Option-l	File name operator	[¬list]	Matches any character not in the list.	[¬A-F]	Match any character that is not in the set A-F.
»	Option-Shift-\	File name operator	«n»	Delimits number standing for number of occurrences.	[X]«2»	Match two occurrences of the character X.
«	Option-\	File name operator	«n»	Delimits number standing for number of occurrences.	[X]«2»	Match two occurrences of the character X.
[	[	File name operator	[...]	Delimits a pattern.	[A-F]	Match any character in the set A–F.
]	]	File name operator	[...]	Delimits a pattern.	[A-F]	Match any character in the set A–F.
≈	≈	File name operator	≈	Matches any number of any characters in a file name.	≈.c	Any file name with the extension ".c".

How File Name Generation Operators Are Used

If an unquoted word in a command contains a file name generation operator, it is considered a file name pattern. When a file name pattern is encountered in a command, MPW replaces the pattern with an alphabetically sorted list of file names that the pattern matches. Then, if the command you are using is one that lists file names—such as Files or Volumes—the generated list is written to standard output. If no file name matches the pattern you have specified, an error is returned.

For example, the command

```
Files ≈
```

works just like the Files command with no parameter; it lists all the files in the current directory. The command

```
Files ≈.c
```

lists all the files in the current directory with names that end with the extension ".c". The command

```
Files Source.?
```

lists every file in the current directory whose name begins with the word "Source" and has a one-letter extension, for instance, the Source.c, Source.p, Source.a, and Source.r files. The command

```
Files Source≈
```

lists every file in the current directory whose name begins with the word "Source," for example, Source, Source.c, Source.p, Source.a, Source.r, Sources, and SourceFile.

To search for file names that match a pattern, you can enclose the pattern in square brackets. For example, the command

```
Files [A-Za-z]+.c
```

lists file names that are made up of uppercase and lowercase letters and are followed by the extension ".c".

File name generation operators can be used with commands that perform operations on files and directories as well as with commands that generate lists. For instance, the command

```
Delete [A-Za-z0-9:]+.p
```

deletes all files in the current directory with names that are made up of letters, digits, and colons, and that end with the extension ".p". The command

```
Catenate ≈.c > MyCSources
```

merges all C source files in the current directory into one file called MyCSources. The command

```
Duplicate ≈.p PascalFolder
```

copies all Pascal files in the current directory into a directory called PascalFolder.

▶ Arithmetic and Logical Operators

The MPW command language has a broad range of operators that can be used to perform arithmetic and logical operations. In MPW, arithmetical and logical operations are always performed either by the command Evaluate or inside conditional loops. Thus, you can use an arithmetical or logical operator as part of an operation performed by the Evaluate command as follows:

```
Evaluate 2+2
```

—or in a conditional loop, like this:

```
If 2+2 == 4
    Beep
End
```

The arithmetical and logical operators used in the MPW shell language are listed in Table 4-8.

Table 4-8. Arithmetic and logical operators

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
!	!	Operator	!n	Not (same as NOT).	Evaluate !0	Output: 1
<>	!= (same as !=, ≠)	Operator	n1 <> n2	True if n1 is not equal to n2.	Evaluate 2<> 3	Output: 1
!=	!= (same as <>, ≠)	Operator	n1 != n2	True if n1 is not equal to n2.	Evaluate 2 != 3	Output: 1

Table 4-8. Arithmetic and logical operators (continued)

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
%	% (same as MOD)	Operator	n1 % n2	Returns mod n2.	Evaluate 25% 4	Output: 1
&	&	Operator	n1 & n2	Bitwise AND.	Evaluate 0b0001 & 0b0011	Output: 1
&&	&&	Operator	n1 && n2	Logical AND.	Evaluate 1 && 1	Output: 1
*	*	Operator	n1 * n2	Multiplies n1 by n2.	Evaluate 3 * 3	Output: 9
+	+	Operator	n1 + n2	Adds n1 to n2.	Evaluate 1 + 1	Output: 2
-	-	Operator	n2 - n1	Subtracts n1 from n2.	Evaluate 33 - 32	Output: 1
<	<	Operator	n1 < n2	True if n1 is less than n2.	Evaluate 2 < 3	Output: 1
<<	<<	Operator	n1 << n2	Shifts n1 left arithmetically n2 times.	Evaluate 0b0001 << 1	Output: 2
<=	<=	Operator	n1 <= n2	True if n1 is less than or equal to n2.	Evaluate 2 <= 3	Output: 1
<=	<= (same as ≤)	Operator	n1 <= n2	True if n1 is less than or equal to n2.	Evaluate 2 <= 3	Output: 1
==	==	Operator	n1 == n2	True if n1 equals n2.	Evaluate 2 == 3	Output: 0
>=	>= (same as ≥)	Operator	n1 >= n2	True if n1 is greater than or equal to n2.	Evaluate 3 >= 2	Output: 1
>>	>>	Operator	n1 >> n2	Shifts n1 right logically n2 times.	Evaluate 0b0010 >> 1	Output: 2
DIV	DIV (same as ÷)	Operator	n1 DIV n2	Divides n1 by n2.	Evaluate 25 DIV 5	Output: 5
MOD	MOD (Same as %)	Operator	n1 MOD n2	Returns mod n2.	Evaluate 25 MOD 4	Output: 1
NOT	NOT	Operator	NOT n	Not (same as !).	Evaluate NOT 0	Output: 1
÷	Option-/ (same as DIV)	Operator	n1 ÷ n2	Divides n1 by n2.	Evaluate 25 ÷ 5	Output: 5

Table 4-8. Arithmetic and logical operators (continued)

<table>
<tr><th>Chr</th><th>Type</th><th>Category</th><th>Syntax</th><th>Meaning</th><th>Example</th><th>Translation</th></tr>
<tr><td>≤</td><td>Option-< (same as <=)</td><td>Operator</td><td>n1 ≤ n2</td><td>True if n1 is less than or equal to n2.</td><td>Evaluate 2 ≤ 3</td><td>Output: 1</td></tr>
<tr><td>≠</td><td>Option-= (same as !=, <>)</td><td>Operator</td><td>n1 ≠ n2</td><td>True if n1 is not equal to n2.</td><td>Evaluate 2 ≠ 3</td><td>Output: 1</td></tr>
<tr><td>≥</td><td>Option-> (same as >=)</td><td>Operator</td><td>n1 ≥ n2</td><td>True if n1 is greater than or equal to n2.</td><td>Evaluate 3 ≥ 2</td><td>Output: 1</td></tr>
<tr><td>^</td><td>^</td><td>Operator</td><td>n1 ^ n2</td><td>Bitwise XOR.</td><td>Evaluate 0b0001 ^ 0b0011</td><td>Output: 2</td></tr>
<tr><td>|</td><td>|</td><td>Operator</td><td>n1 | n2</td><td>Bitwise AND.</td><td>Evaluate 0b0001 | 0b0011</td><td>Output: 3</td></tr>
<tr><td>||</td><td>||</td><td>Operator</td><td>n1 || n2</td><td>Logical OR.</td><td>Evaluate 1 || 0</td><td>Output: 1</td></tr>
<tr><td>~</td><td>~</td><td>Operator</td><td>~n</td><td>Negates number.</td><td>Evaluate ~ 4</td><td>Output: -5</td></tr>
</table>

Important ▶

Double Meaning Department. Many of the characters in Table 4-8 are also used in regular expressions—and have completely different meanings. For example, the | is a pipe character when used as a regular expression operator, but it is a bitwise OR operator when used as a logical operator. The + and * characters are used as counters in regular expressions, but they are addition and multiplication operators when used in arithmetic operations. The < and > characters, which mean "less than" and "greater than" in logical operations, are redirection operations when they are used with commands.

To avoid getting confused by these double meanings, just remember that when a special character occurs in an arithmetic or logical operation that is part of an Evaluate command, it is always an arithmetic or logical operator. If it occurs anywhere else, it is not.

Arithmetic Operators

The MPW command language uses these arithmetic operators:

Operator	*Meaning*
+	Addition operator
-	Subtraction operator
*	Multiplication operator
÷ (Option-/)	Division operator
DIV	Division operator (alternate)
%	Modula
MOD	Modula (alternate)

In the MPW command language, arithmetic expressions always follow the Evaluate command. For example, the command

```
Evaluate {a} + {b}
```

adds the variables {a} and {b}.

To make use of the output of the Evaluate command, the command substitution operator (`) is often used to convert Evaluate's output into a parameter of another command. For example, the command

```
Echo `Evaluate {a} + {b}`
```

echoes the result of the expression {a} + {b} (the use of the word Echo is optional). And the command

```
Set x `Evaluate {a} + {b}`
```

sets the variable {x} to the sum of {a} + {b}.

Logical and Shift Operators

The logical, shift-left, and shift-right operators used in the MPW command language are as follows.

Operator	*Meaning*
==	Equal to
!=	Not equal to
<>	Not equal to
≠	Not equal to
&&	AND
||	OR
<	Less than
>	Greater than
<=	Less than or equal to
≤ (Option-<)	Less than or equal to
>=	Greater than or equal to
≥ (Option->)	Greater than or equal to
!	Logical negation
¬ (Option-l)	Logical negation
NOT	Logical negation
<<	Arithmetic shift left
>>	Logical shift right

Logical operations, like arithmetic operations, must follow the Evaluate command on a command line. For instance, the command

```
Set x `Evaluate {a} == {b}`
```

sets the value of the variable x to 1, or true, if variable {a} is equal to variable {b}, and to 0, or false, if variable {a} is not equal to variable {b}.

For more examples of logical operations in MPW, see Listing 2-2 and the section on the Evaluate command in Chapter 2.

Number Prefixes

The Evaluate command works with decimal, hexadecimal, octal, and binary numbers. You can designate the kind of number you are using in an Evaluate operation by using a number prefix. The number prefixes used in the shell language are listed in Table 4-9.

Table 4-9. Number prefixes

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
$	$	No. prefix	$[0-9A-Fa-f]+	Precedes hexadecimal number (same as 0x).	Evaluate $9EFF + $9E	Output: 40861
0	0 (zero)	No. prefix	0[0-7]+	Precedes octal number.	Evaluate 054 + 030	Output: 68
0b	0b	No. prefix	0b[0-1]+	Precedes binary number.	Evaluate 0b11 - 0b01	Output: 2
0x	0x	No. prefix	$[0-9A-Fa-f]	Precedes hexadecimal number (same as $).	Evaluate $9EFF + $9E	Output: 40861

To designate the base of a number in an Evaluate operation, all you have to do is precede the number with the appropriate number prefix. For example, the commands

```
Evalute $4B5E + $7D80
```

and

```
Evalute 0x4B5E + 0x7D80
```

both add the hexadecimal numbers 4B5E and 7D80.

Redirection Operators

Redirection operators are characters that can be used to redirect the output of a command to a specified file or set of files.

For example, the command

```
Echo "A new beginning" > "{Target}"
```

redirects the output of the Echo command to the target window (ordinarily, the active window is the destination of text written by the Echo command).

Note that in this example the string "A new beginning" replaces all the text in the document in the target window. That is the function of the redirection operator >.

To append text to a document, instead of replacing the text in the document, you must use the redirection operator >>, like this:

```
Echo "End of file" >> "{Target}"
```

You can use redirection operators to redirect the diagnostic output of MPW commands as well as to redirect text. Detailed information on how redirection operators are used in MPW was presented in Chapter 2. The redirection operators used in the MPW command language are listed in Table 4-10.

Table 4-10. Redirection operators

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
<	<	Redirection	<f	Standard input is taken from file name f.	Alert < Errors	Display an alert dialog containing the contents of the file Errors.
>	>	Redirection	>f	Redirects standard output, replacing contents of file f.	Echo "{Status}" > Errors	Write contents of shell variable {Status} to file Errors, replacing its previous contents.
>>	>>	Redirection	>>f	Redirects standard output, appending it to contents of file f.	Echo "{Status}" >> Errors	Append contents of shell variable {Status} to the end of file Errors.
≥	Option->	Redirection	≥f	Redirects diagnostics, replacing contents of file f.	(Files ≈.p | |) ≥ Errors	List file.names that end in ".p". Send diagnostics to file Errors, replacing its contents.
≥≥	Option->	Redirection	≥≥f	Redirects diagnostics; append to contents of file f.	(Files ≈.p | |) ≥ ≥Errors	List file names that end in ".p". Append diagnostics to end of file Errors.
Σ	Option-W	Redirection	Σf	Redirects both standard output and diagnostics to file f, replacing its contents.	(Files ≈.p | |) Σ Temp	List file names ending in ".p". Send output, diagnostics to file Temp, replacing its contents.
ΣΣ	Option-W	Redirection	ΣΣf	Redirects both standard output and diagnostics to file f; append to file f.	(Files ≈.p | |) ΣΣ Temp	List file names ending in ".p". Append output and diagnostics to file Temp.

▶ Special Characters Used in Menus

One set of special characters, called metacharacters, are used only with the AddMenu command. The metacharacters used with the AddMenu command are listed in Table 4-11. For a detailed explanation about how these characters are used, see the section on the AddMenu command in Chapter 3.

Table 4-11. Metacharacters used by the AddMenu command

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
!	!	Menus	!c	Marks menu item with specified character.	!√	Mark menu item with a check mark.
(	(	Menus	(	Disables menu item.	(-	Place a dimmed horizontal line in menu list.
-	-	Menus	-	Prints a horizontal line separating menu items.	(-	Place a dimmed horizontal line in menu list.
/	/	Menus	/c	Associates a menu item with keyboard equivalent c.	/M	Assign control-M to be menu item's keyboard equivalent.
<	<	Menus	<[BIUOS]	Sets character style of a menu item (bold, italics, underlined, outline, or shadow).	<B	Set character style of menu item to bold.
^	^	Menus	^n	Followed by an icon number, marks menu item with specified icon.	^2	Mark the menu item with icon no. 2 (in a resource fork).

▶ Special Characters Used in Makefiles

The special characters *ƒ* (Option-f) and *ƒƒ* are used only in makefiles: scripts used to build programs, or convert them from source code into executable programs. Several other special characters are also used in makefile scripts.

Makefiles have their own special language, and it is a little different from the MPW command language. Makefiles and the makefile language are described in detail in Chapter 5. Table 4-12 lists the special characters that are used in the MPW makefile language.

Table 4-12. Special characters used in makefiles

Chr	*Type*	*Category*	*Syntax*	*Meaning*	*Example*	*Translation*
"	"	Make	"s"	Delimits a string in which each character is taken literally, except for ∂, {}, and `.	"{Libraries}" Runtime.o	The runtime libraries.
#	#	Make	#s	Interprets characters between # and terminator as a comment.	### Dependency rules ###	Interpret string following # as a comment.
'	'	Make	's'	Delimits a string in which all characters are taken literally.	'{Libraries}' Runtime.o	The runtime libraries.
∂	Option-D	Make	∂	If ∂ stands alone at end of a line, MPW joins line to next line, ignoring return.	(First line:) Sample *ƒƒ* Sample.p.o ∂ (Second line:) Sample.r	Output: Sample *ƒƒ* Sample.p.o Sample.r.
ƒ	Option-F	Make	f1 *ƒ* f1	File f1 depends on file f2.	Sample.p.o *ƒ* Sample.p	File Sample.p.o depends on file Sample.p.
ƒƒ	Option-F	Make	f1 *ƒƒ* f1	File f1 depends on file f2, and f2 has its own build commands.	Sample *ƒƒ* Sample.p.o	File Sample depends on file Sample.p.o, and Sample.p.o. has its own set of build commands.

▶ Using Special Characters in Scripts

Now that you have been introduced to the special characters used in the MPW command language, we can take a closer look at a code fragment and two short scripts that were presented in earlier chapters, along with promises that they would be decoded. If you have carefully studied Chapters 2, 3, and 4, you may now be able to understand how they work.

First, we will examine a fragment of code that was presented in Chapter 2. It is the code in the MPW UserStartup script that executes any supplementary UserStartup file with a name written in the format

```
UserStartup•filename
```

This is the code:

```
For __Startup__i In `(Files "{ShellDirectory}" ∂
UserStartup•≈ || Set Status 0) ≥ dev:null`
Execute "{__Startup__i}"
End
Unset __Startup__i
```

This fragment of code uses a For...In loop, a structured construct described in Chapter 2. In the first line, it defines a variable named __Startup__i and uses the command substitution operator (`...`) to determine the name of every file in the MPW directory that has a name written in the format UserStartup•*filename*.

If no such file is found, the conditional command terminator || sets the shell variable {Status} (in which errors are returned) to 0—fooling MPW into thinking there was no error. Then it uses the ≥ operator to discard any error messages by redirecting them to the pseudo-device Dev:Null—the "bit bucket."

During the For loop, the {__Startup__i} variable is set to the name of each UserStartup•≈ file that is found, and the file is executed. When all such files have been executed, the {__Startup__i} variable is Unset, or discarded.

The next example is a login script that was introduced in Chapter 2 as Listing 3-2. In this incarnation, we'll call it Listing 4-2.

Listing 4-2. Login script revisited

```
Set Exit 0
Set N "`Request 'Please type the password:'`"
If "{Status}" == 0
    If "{N}" =~ /Karoshi/
        # This happens if login succeeds
        Chimes
        Alert -s "Welcome to the Editor!"
        New
        Echo "You are now online.∂n" > "{Active}"
    Else
        # This happens if login fails
        Alert "Password invalid; access denied."
    End
End
Set Exit 1
```

This is an easy one. In the first line, it resets the shell variable {Exit} to 0, so that any error generated by the script will not terminate the script. (Normally, the {Exit} variable is set to 1, and any scripts that result in errors come to an abrupt halt.)

In the second line, the script defines a variable named {N}. Then it uses the Request command to display a request dialog displaying the "Please type the password:" prompt. When the user types in a password, the command substitution operator (`...`) is used to set the {N} variable to the password input by the user.

If the {N} variable is assigned the value "Karoshi"—the correct password—the script executes another script, named Chimes, which plays a melody. (Chimes, you may recall, is a Beep script created in Chapter 2.) The script then displays an alert dialog that proclaims, "Welcome to the Editor!" Finally, it opens a new window and displays there the message, "You are now online."

If the user fails to type in the correct password, the script displays an alert that says, "Password invalid; access denied."

Our final example was called Listing 3-3 back in Chapter 3. Here it is revived as Listing 4-3. This listing was used to streamline the operation of the UserVariables Commando when the commando was introduced in Chapter 3.

In case you have forgotten, it lets you use the UserVariables commando to redefine any shell variables that you want to change, and then adds them automatically to your UserStartup script.

Without the script, the UserVariables commando merely writes the new values you have set to the active window. Then, if you want to add them to your UserStartup script, you have to open your UserStartup script and paste them in using editing commands.

With the script, all you have to do is invoke the UserVariables commando and select new values for any shell variables that you want to redefine. When you close the commando, they are appended to your UserStartup script automatically.

Listing 4-3. UserVar script

```
# UserVar Script
#
# Writes the output of the UserVariables Commando
# To your UserStartup Script
#
# Just Execute the Command UserVariables
# And run this script;
# It will do the rest
```

```
Find ∞ "{Active}"   # Go to end of window
## Search backwards for output of UserVariables
Find \UserVariables ; \Δ "{Active}"
Find §:∞ "{Active}"    #Select it
## Open UserStartup script
Open "{ShellVariable}"UserStartup
Find ∞ "{ShellVariable}"UserStartup    # Go to end
## Write a return to start a new line
Echo "∂n" >> "{ShellVariable}"UserStartup
## Add output of UserVariables to UserStartup
Catenate "{Target}".§ >> "{ShellVariable}"UserStartup
```

This script looks a lot more complicated than it is. In the first line, it places the insertion point at the end of the file in the active window. Then it uses the Find command to search backwards for the end of the string "UserVariables ; ", which the UserVariables command has written into the active window.

In the next line, the script selects the string "UserVariables ; ". Then it opens the UserStartup script, goes to the end of the script, and writes a newline character (∂n). Then it adds any new variable definitions that the UserVaribles commando has written to the UserStartup script.

▶ The Special Characters at a Glance

Table 4-13 is a listing of the special characters used in MPW, arranged in character order.

By the Way ▶

Here, for the First Time Anywhere .... Table 4-13 is the first listing of its kind that has appeared in any book about MPW. Nowhere in the *MPW 3.0 Reference*—or in any other MPW book—can you find a table, a listing, or a segment of text that lists and describes every special character used in the MPW command language. In fact, the *MPW 3.0 Reference* does not even tell how many kinds of special characters there are in MPW—and neither does any other book about MPW that I've ever seen. So, when I started writing this chapter, I took the liberty of drawing up my own list of categories.

According to my count, the special characters used in the MPW shell language can be broken down into fifteen categories. But these categories overlap each other in various ways. So if you sat down with a blank piece of paper and made a list of MPW's special characters, your list of categories might be different from mine.

Table 4-13. The MPW special character set

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
!	!	Menus	Marks menu item with specified character.	!c	!√	Mark menu item with a check mark.
!	!	Operator	Not (same as NOT).	!n	Evaluate !0	Output: 1
!	!	Selection	Selects the line that is *n* lines after end of current selection.	!n	Find !3	Select the third line after the current selection.
!	!	Selection	Places insertion point *n* characters after regular expression.	r!n	Find /alpha/!3	Place insertion point three characters after the word "alpha."
<>	!= (same as !=, ≠)	Operator	True if n1 is not equal to n2.	n1 <> n2	Evaluate 2 <> 3	Output: 1
!=	!= (same as <>, ≠)	Operator	True if n1 is not equal to n2.	n1 != n2	Evaluate 2 != 3	Output: 1
!~	!~	Regular expression operator	True if s1 is not equal to s2.	"s1" !~ /s2/	Evaluate "alpha" !~ /beta/	Output: 1
"	"	Delimiter	Delimits a string in which each character is taken literally, except for ∂, {}, and `.	"s"	Print "{MPW}" >> "{Target}"	Echo the contents of the shell variable {MPW} to the target window.
"	"	Make	Delimits a string in which each character is taken literally, except for ∂, {}, and `.	"s"	"{Libraries}" Runtime.o	The runtime libraries.
#	#	Comment	Interprets characters between # and terminator as a comment.	#s	# This won't work	Interpret string following # as a comment.
#	#	Make	Interprets characters between # and terminator as a comment.	#s	### Dependency rules ###	Interpret string following # as a comment.
$	$	No. prefix	Precedes hexadecimal number (same as 0x).	$[0-9A-Fa-f]+	Evaluate $9EFF + $9E	Output: 40861

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
%	% (same as MOD)	Operator	Returns mod n2.	n1 % n2	Evaluate 25 % 4	Output: 1
&	&	Operator	Bitwise AND	n1 & n2	Evaluate 0b0001 & 0b0011	Output: 1
&&	&&	Operator	Logical AND	n1 && n2	Evaluate 1 && 1	Output: 1
&&	&&	Terminator	Executes c2 command if c1 command succeeds.	c1 && c2	Find /charlie/ && Echo Found!	If string "charlie" is found, MPW echoes, "Found!"
'	'	Delimiter	Delimits a string in which all characters are taken literally.	's'	Echo '{MPW}' >> "{Target}"	Echo the string "{MPW}" to the target window.
'	'	Make	Delimits a string in which all characters are taken literally.	's'	'{Libraries}' Runtime.o	The runtime libraries.
(	(	Delimiter	Delimits a group of characters that form a pattern.	(p)	Find /("*")+/	Select a group of one or more asterisks.
(	(	Menus	Disables menu item.	(	(-	Place a dimmed horizontal line in menu list.
)	)	Delimiter	Delimits a group of characters that form a pattern.	(p)	Find /("*")+/	Select a group of one or more asterisks.
*	*	File name operator	Matches zero or more occurrences of the preceding character or character list.	c*	X*	Match zero or more occurrences of the character X.
*	*	Operator	Multiplies n1 by n2.	n1 * n2	Evaluate 3 * 3	Output: 9

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
*	*	Regular expression operator	Selects zero or more occurrences of regular expression.	r*	Find /('*')+'/' [∂r∂t]*/	Select a group of one or more asterisks followed by a slash bar and 0 or more white spaces.
+	+	File name operator	Matches one or more occurrences of the preceding character or characters.	c+	X+	Match one or more occurrences of the character X.
+	+	Operator	Adds n1 to n2.	n1 + n2	Evaluate 1 + 1	Output: 2
+	+	Regular expression operator	Selects one or more occurrences of regular expression.	r+	Find /('*')/	Select a group of one or more asterisks.
+	+	Regular expression opeartor	Matches one or more occurrences of the preceding character or characters.	r+	X+	Match one or more occurrences of the character X.
-	-	Menus	Prints a horizontal line separating menu items.	-	(-	Place a dimmed horizontal line in menu list.
-	-	Operator	Subtracts n1 from n2.	n2 - n1	Evaluate 33 - 32	Ouput: 1
-	-	Regular expression operator	Stands for range of characters between c1 and c2.	c1-c2	Find /[A-Za-z] +∂n/	Select any word made up of upper- and lower-case letters that appears at the end of a line.
/	/	Delimiter	Searches forward and selects regular expression.	/r/	Find /delta/	Search forward and select the word "delta."
/	/	Menus	Associates a menu item with keyboard equivalent c.	/c	/M	Assign control-M to be menu item's keyboard equivalent.

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
0	0 (zero)	No. prefix	Precedes octal number.	0[0-7]+	Evaluate 054 + 030	Output: 68
0b	0b	No. prefix	Precedes binary number.	0b[0-1]+	Evaluate 0b11 - 0b01	Output: 2
0x	0x	No. prefix	Precedes hexadecimal number (same as $).	$[0-9A-Fa-f]	Evaluate $9EFF + $9E	Output: 40861
;	;	Terminator	Treats commands on the same line as if they were on different lines.	c ; c	Echo hello ; Echo goodbye	Output: (First line:) Hello (Second line:) Goodbye
<	<	Menus	Sets character style of a menu item (bold, italics, underlined, outline, or shadow).	<[BIUOS]	<B	Set character style of menu item to bold.
<	<	Operator	True if n1 is less than n2.	n1 < n2	Evaluate 2 < 3	Output: 1
<	<	Re-direction	Standard input is taken from file name f.	<f	Alert < Errors	Display an alert dialog containing the contents of the file Errors.
<<	<<	Operator	Shift n1 left arithmetically n2 times.	n1 << n2	Evaluate 0b0001 << 1	Output: 2
<=	<=	Operator	True if n1 is less than or equal to n2.	n1 <= n2	Evaluate 2 <= 3	Output: 1
<=	<= (same as ≤)	Operator	True if n1 is less than or equal to n2.	n1 <= n2	Evaluate 2 <= 3	Output: 1
==	==	Operator	True if n1 equals n2.	n1 = = n2	Evaluate 2 == 3	Output: 0

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
=~	=~	Regular expression operator	True if s1 equals s2.	"s1" =~ /s2/	Evaluate "beta" =~ /beta/	Output: 1
>	>	Re-direction	Redirect standard output, replacing contents of file f.	>f	Echo "{Status}" > Errors	Write contents of shell variable {Status} to file Errors, replacing its previous contents.
>=	>= (same as ≥)	Operator	True if n1 is greater than or equal to n2.	n1 >= n2	Evaluate 3 >= 2	Output: 1
>>	>>	Operator	Shift n1 right logically n2 times.	n1 >> n2	Evaluate 0b0010 >> 1	Output:2
>>	>>	Re-direction	Redirect standard output, appending it to contents of file f.	>>f	Echo "{Status}" >> Errors	Append contents of shell variable {Status} to the end of file Errors.
?	?	Filename operator	Matches any single character in a file name.	?	Source.?	Any file that is named Source and has a one-character extension.
?	?	Wildcard	Matches any single character in a string.	?	Find /Bar?/	Select any four-character word that begins with "Bar."
?*	?*	Wildcard	Matches any number of occurrences of any character (same as ≈).	chars?*	Find /Mar?*/	Select any word that begins with "Mar."
?*	?* (same as ≈)	Filename operator	Matches any number of any characters in a file name.	?*	?*.c	Any file name with the extension ".c."
:	Colon	Regular expression opeartor	All text between (two selections).	s:s	Find •:∞	Select (highlight) all text in file.
DIV	DIV (same as ÷)	Operator	Divides n1 by n2.	n1 DIV n2	Evaluate 25 DIV 5	Output: 5

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
MOD	MOD (Same as %)	Operator	Returns mod n2.	n1 MOD n2	Evaluate 25 MOD 4	Output: 1
NOT	NOT	Operator	Not (same as !).	NOT n	Evaluate NOT 0	Output: 1
¡	Option-!	Selection	Places insertion point *n* lines before start of current selection.	¡n	Find ¡3	Place insertion point three lines before start of current selection.
÷	Option-/ (same as DIV)	Operator	Divides n1 by n2.	n1 ÷ n2	Evaluate 25 ÷ 5	Output: 5
∞	Option-5	Regular expression operator	(With command that takes a -c option): Repeats command to end of file.	cmd -c ∞	Replace -c ∞ /123/ 456	Replace string "123" with string "456" every time it appears in target window.
∞	Option-5	Regular expression operator	Selects regular expression at the end of a line	r∞	Find /arlie∞/	Select the letters "arlie" at the end of a line.
∞	Option-5	Selection	Selects end of file.	∞	Find ∞	Place insertion point after last character in file.
§	Option-6	Selection	Current selection.	§	Copy §	Copy the current selection (high-lighted text) to the Clipboard.
•	Option-8	Regular expression operator	Selects regular expression at the beginning of a line.	•r	Find /•ch/	Select the letters "ch" at the beginning of a line.
•	Option-8	Selection	Selects beginning of file.	•	Find •	Place insertion point before first character in file line.
…	Option-;	Regular expression operator	Executes Commando command, invokes Commando dialog for command c.	c…	TileWin-dows…	Invoke Tile Windows Commando.

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
≤	Option-< (same as <=)	Operator	True if n1 is less than or equal to n2.	n1 ≤ n2	Evaluate 2 ≤ 3	Output: 1
≠	Option- = (same as !=, <>)	Operator	True if n1 is not equal to n2.	n1 ≠ n2	Evaluate 2 ≠ 3	Output 1
≥	Option->	Re-direction	Redirect diagnostics, replacing contents of file f.	≥f	(Files ≈.p | |) ≥ Errors	List file names that end in ".p". Send diagnostics to file Errors, replacing its contents.
≥≥	Option->	Re-direction	Redirect and append diagnostics to file f.	≥≥f	(Files ≈.p | |) ≥ ≥ Errors	List file names that end in ".p". Append diagnostics to end of file Errors.
≥	Option-> (same as >=)	Operator	True if n1 is greater than or equal to n2.	n1 ≥ n2	Evaluate 3 ≥ 2	Output: 1
∂	Option-d	Escape	Return.	∂n	Echo ∂n	Echo a return.
∂	Option-d	Escape	Tab.	∂t	Echo ∂n	Echo a tab.
∂	Option-d	Escape	Form feed.	∂f	Echo ∂n	Echo a form feed.
∂	Option-d	Escape	Defeats the meaning of the special character that follows it.	∂¬	Echo ∂¬	Output: ¬
∂	Option-d	Line continua-tion	If ∂ stands alone at end of a line, MPW joins line to next line, ignoring return.	l ∂ l	(First line:) Echo "How are ∂ (Second line:) you today?"	Output: How are you today? (All on one line)
∂	Option-d	Make	If ∂ stands alone at end of a line, MPW joins line to next line, ignoring return.	l ∂ l	(First line:) Sample *ƒƒ* Sample.p.o ∂ (Second line:) Sample.r	Output: Sample *ƒƒ* Sample.p.o Sample.r
ƒ	Option-f	Make	File f1 depends on file f2.	f1 *ƒ* f1	Sample.p.o *ƒ* Sample.p	File Sample.p.o depends on file Sample.p.

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
ƒƒ	Option-f	Make	File f1 depends on file f2, and f2 has its own build commands.	f1 *ƒƒ* f1	Sample *ƒƒ* Sample.p.o	File Sample depends on file Sample.p.o, and Sample p.o. has its own set of build commands.
Δ	Option-j	Selection	Places insertion point before first character in regular expression.	Δr	Find Δ/charlie/	Place insertion point before first character in the word "charlie."
Δ	Option-j	Selection	Places insertion point after last character of regular expression.	rΔ	Find /charlie/Δ	Place insertion point after last character of the word "charlie."
¬	Option-l	File name operator	Matches any character not in the list.	[¬list]	[¬A-F]	Match any character that is not in the set A–F.
¬	Option-l	Regular expression operator	Any character not in the list.	[¬list]	Replace -c ∞ /[¬A-Za-z∂n" "]/ "*"	Replace all characters except A–Z, a–z, returns and spaces with asterisks.
®	Option-r	Regular expression operator	Tags regular expression with a number (range: 1–9).	r®n	Replace /([a-zA-Z]+)®1[ ]+([a-zA-Z]+)®2/ '®2 ®1'	Reverse the order of two words separated by one or more spaces.
»	Option-Shift-\	Delimiter	Delimits number standing for number of occurrences.	«n»	Find /[∂t]«2»/	Select exactly two tabs.
»	Option-Shift-\	Delimiter	Delimits number standing for at least *n* occurrences.	«n,»	Find /[∂t]«2,»/	Select two or more tabs.
»	Option-Shift-\	Delimiter	Delimits number standing for *n* to *n* occurrences.	«n1,n2»	Find /[∂t]«2,4»	Select two to four tabs.
»	Option-Shift-\	File name operator	Delimits number standing for number of occurrences.	«n»	[X]«2»	Match two occurrences of the character X.

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
Σ	Option-w	Re-direction	Redirects both standard output and diagnostics to file f, replacing its contents.	Σf	(Files ≈.p ∣∣) Σ Temp	List file names ending in ".p". Send output, diagnostics to file Temp, replacing its contents.
ΣΣ	Option-w	Re-direction	Redirects and appends both standard output and diagnostics to file f.	ΣΣf	(Files ≈.p ∣∣) ΣΣ Temp	List file names ending in ".p". Append output and diagnostics to file Temp.
≈	Option-x	Wildcard	Matches any number of any characters in a string.	≈	Find /Mar≈/	Select any word that begins with "Mar."
«	Option-\	Delimiter	Delimits number standing for number of occurrences.	«n»	Find /[∂t]«2»/	Select exactly two tabs.
«	Option-\	Delimiter	Delimits number standing for at least *n* occurrences.	«n,»	Find /[∂t]«2,»/	Select two or more tabs.
«	Option-\	Delimiter	Delimits number standing for *n* to *n* occurrences.	«n1,n2»	Find /[∂t]«2,4»/	Select two to four tabs.
«	Option-\	Filename operator	Delimits number standing for number of occurrences.	«n»	[X]«2»	Match two occurrences of the character X.
Return	Ret	Terminator	Ends command.	c (r)	Echo Hello(r)	Output: Hello
Space	Space	White space	Separates words.	w w	Echo Hello	Output: Hello
Tab	Tab	White space	Separates words.	w w	Echo Hello	Output: Hello
[	[	Delimiter	Delimits a pattern.	[...]	Find /[A-F]/	Search for any character in the set A–F
[	[	File name operator	Delimits a pattern.	[...]	[A-F]	Match any character in the set A–F.
\	\	Delimiter	Searches backwards and selects regular expression.	\r\	Find \alpha\	Search backward and select the word "alpha."

Table 4-13. The MPW special character set (continued)

<table>
<tr><th>Chr</th><th>Press</th><th>Category</th><th>Meaning</th><th>Usage</th><th>Example</th><th>Translation</th></tr>
<tr><td>]</td><td>]</td><td>Delimiter</td><td>Delimits a pattern.</td><td>[...]</td><td>Find /[A-F]/</td><td>Search for any character in the set A–F.</td></tr>
<tr><td>]</td><td>]</td><td>File name operator</td><td>Delimits a pattern.</td><td>[...]</td><td>[A-F]</td><td>Match any character in the set A–F.</td></tr>
<tr><td>^</td><td>^</td><td>Menus</td><td>Followed by an icon number, marks menu item with specified icon.</td><td>^n</td><td>^2</td><td>Mark the menu item with icon no. 2 (in a resource fork).</td></tr>
<tr><td>^</td><td>^</td><td>Operator</td><td>Bitwise XOR.</td><td>n1 ^ n2</td><td>Evaluate 0b0001 ^ 0b0011</td><td>Output: 2</td></tr>
<tr><td>`</td><td>`</td><td>Delimiter</td><td>Send output of c2 command to c1 command for processing.</td><td>c1 `c2`</td><td>Echo `Files -t TEXT`</td><td>Files command sends its output to Echo command, which prints the output on the screen.</td></tr>
<tr><td>{</td><td>{</td><td>Delimiter</td><td>Delimits variable v.</td><td>{v}</td><td>Echo "{MPW}"</td><td>Echo contents of shell variable {MPW}.</td></tr>
<tr><td>|</td><td>|</td><td>Operator</td><td>Bitwise AND.</td><td>n1 | n2</td><td>Evaluate 0b0001 | 0b0011</td><td>Output: 3</td></tr>
<tr><td>|</td><td>|</td><td>Terminator</td><td>Pipes output of c1 command to input of c2.</td><td>c1 | c2</td><td>Files | Count -l</td><td>Files pipes a list of files to Count, which prints the list on the screen.</td></tr>
<tr><td>||</td><td>||</td><td>Operator</td><td>Logical OR.</td><td>n1 || n2</td><td>Evaluate 1 || 0</td><td>Output: 1</td></tr>
<tr><td>||</td><td>||</td><td>Terminator</td><td>Executes c2 command if c1 command fails.</td><td>c1 || c2</td><td>Find /zebra/ || Echo Sorry!</td><td>Search for string "zebra" and echo "Sorry!" if search fails.</td></tr>
</table>

Table 4-13. The MPW special character set (continued)

Chr	*Press*	*Category*	*Meaning*	*Usage*	*Example*	*Translation*
}	}	Delimiter	Delimits variable v.	{v}	Echo "{MPW}"	Echo contents of shell variable {MPW}.
~	~	Operator	Negates number.	~n	Evaluate ~ 4	Output: -5
≈	≈	Filename operator	Matches any number of any characters in a file name.	≈	≈.c	Match any file name with the extension ".c."

▶ Conclusion

This chapter concludes Part I of this book—the part that focuses on the basic concepts of the MPW command language. Now the fun begins.

In Part II, you'll learn more about Big Three Macintosh managers—the Event Manager, the Resource Manager, and the Memory Manager—and you'll have an opportunity to write an application program, an MPW tool, and a desk accessory. Then, in the final chapter, you'll learn some important MPW programming secrets, and start doing some real power programming.

PART TWO

▶ Writing an Application

In Part 1, we examined the Macintosh and the Macintosh Programmer's Workshop, and you learned how MPW and the Macintosh work together. In Part 2, you will get a chance to put this knowledge to use in a practical way: by writing, compiling, building, and executing a Macintosh application written using MPW.

Part 2, like Part 1, contains four chapters.

Chapter 5, "Event-Driven Programming," focuses on the Toolbox Event Manager, and shows how to write event-driven programs for the Macintosh using the Macintosh Programmer's Workshop. Particular emphasis is given to writing programs that are compatible with System Software Version 7.0, which uses some event types that were not supported in previous systems.

Chapter 6, "MPW and the Resource Manager," shows and tells how resources are handled in programs written under MPW. It also describes some resources that were added with System Software Version 7.0, and tells how some other resources were changed with the introduction of System 7.0.

Chapter 7, "MPW and the Memory Manager," takes a close look at the memory architecture of the Macintosh, and explains how memory is managed in programs written for System 7.0. Again, there have been some important changes.

Finally, in Chapter 8, "Building an Application," you'll have an opportunity to compile, build, and execute a Macintosh program using MPW.

5 ▸ Event-Driven Programming

If you have read Part I and have tried out most of the examples presented in Chapters 1 through 4, you now know more about MPW than some professional programmers who use it every day. In Part II, you'll have an opportunity to put your knowledge to use, and actually write some programs using MPW.

This chapter shows you how to write an event-driven program—the kind of program that makes use of windows, pull-down menus, and all of the other special features of the Macintosh user interface. This chapter will prove especially helpful if you want to write up-to-date programs that take advantage of the special capabilities of System 7 and the System 6 MultiFinder.

▸ MPW and the Event Manager

As pointed out in Chapter 1, programs written for the Macintosh are very different from programs written for more conventional computers. While old-fashioned computers force the user to navigate through multiple layers of menus and to issue instructions by typing commands, the Macintosh does away with all that nonsense and puts the user in the driver's seat. In a program written for the Macintosh, the user can control everything that a program does, at any time, by opening windows, pulling down menus, clicking on controls, and dragging icons, pictures, and text.

Because Macintosh programs are designed to be so easy on the user, they require a different kind of structure than other kinds of programs do. In technical terms, conventional programs are sometimes referred to as sequential programs, whereas Macintosh-style programs are called event-driven programs.

The main feature of an event-driven program is a loop called the main event loop. The primary task of the main event loop is to look continually for any event—such as a mouse click, the pressing of a key, or the insertion of disk. When an event is detected, the main event loop temporarily passes control of the computer to some other part of the program that is responsible for handling the kind of event that has occurred. That part of the program handles the event and then returns control of the computer back to the main event loop. This process continues until the user quits the application.

▶ How Applications Detect Events

The Macintosh has two managers that detect and handle events. One is called the Toolbox Event Manager. The other is called the Operating System Event Manager.

Ordinarily, programs find out about events by calling the Toolbox Event Manager. But in some cases—for example, when an application defines its own customized kinds of events—the program can also make calls directly to the Operating System Event Manager.

Note ▶

A Tale of Two Managers. Since applications often call the Toolbox Event Manager and rarely call the Operating System Event Manager, the Toolbox Event Manager is often referred to simply as the Event Manager. So, when you see a reference to the Event Manager in this book—or in just about any other book about Macintosh programming—the manager being spoken of is always the Toolbox Event Manager.

The Operating System Event Manager operates at a lower level than does the Toolbox Event Manager; the OS Event Manager keeps track of events waiting to be processed and reports them at the appropriate times to the Toolbox Event Manager. Once the OS Event Manager has reported an event to the Toolbox Event Manager, the Toolbox Event Manager can report it to the application being executed.

▶ Calling the Toolbox Event Manager

Until the introduction of MultiFinder, the call made most often to the Toolbox Event Manager was GetNextEvent. However, programs that are designed to take advantage of the background-processing capabilities of MultiFinder and the multitasking capabilities of System 7 also use the Toolbox Event Manager call WaitNextEvent.

By calling either GetNextEvent or WaitNextEvent, an application can find out when an event has occurred, what kind of event it was, and where the mouse cursor was on the screen when the event was detected. The program can then process the event in any way the author of the program desires.

The difference between the two calls is that WaitNextEvent is used by systems running background tasks; that is, by systems that support, and are currently running, either System 7 or the pre-System 7 MultiFinder. GetNextEvent does not support background processing; it is a call that was designed before MultiFinder became available.

When you want to write an application that is compatible with any Macintosh system—whether it supports background processing or not—you must provide the program with a means of determining what kind of system is running. Then you must write a main event loop that can call either GetNextEvent or WaitNextEvent, depending on what kind of system is running. If you find that System 7 or MultiFinder is in use, your program's main event loop can call WaitNextEvent. Otherwise, it can call GetNextEvent. Procedures for writing a main event loop that is compatible with any system are described later in this chapter.

Important ▶

Preparing to Use the Event Manager. If you want to use the Event Manager to detect window events, you must initialize the Window Manager in the initialization section of your program. However, before you can initialize the Window Manager, you must initialize both QuickDraw and the Font Manager. The procedures for initializing Toolbox managers were explained in Chapter 1.

▶ How Applications Process Events

Applications ordinarily find out what they need to know about events by calling either WaitNextEvent or GetNextEvent. When the user of an application presses the mouse button, types a key on the keyboard or keypad, or inserts a disk in a disk drive, the application generally detects the event by calling WaitNextEvent or GetNextEvent, and then it responds to the event in whatever way is appropriate.

Listing 5-1 is a block of pseudo-code that illustrates how a main event loop might work in a Macintosh application. First, the pseudo-code calls WaitNextEvent, which appears in the third line of the listing. Three lines later, the GetNextEvent call is made.

Listing 5-1. Pseudo-code for a main event loop

```
REPEAT
   IF gHasWaitNextEvent
      (Call WaitNextEvent)
   ELSE
      (Call SystemTask)
      (Call GetNextEvent);

   IF (an event is detected)
      CASE mouse event
         (Call FindWindow)
      CASE menu event
         (Call menu routine)
      CASE desktop event
         (Call SystemClick)
      CASE window event
         (Call window routine)
      CASE control event
         (Call control routine)
      CASE key-down event
         (Call keyboard routine)
      CASE auto-key event
         (Call keyboard or auto-key routine)
      CASE activate event
         (Activate or deactivate window)
      CASE update event
         (Update window)
      CASE OS event (suspend/resume or mouse-moved event)
         (Handle OS event)
      CASE null event
         (Handle null event)
   END;
UNTIL quit
```

▶ The gHasWaitNextEvent Variable

Note that before WaitNextEvent is called in Listing 5-1, the value of a global variable called gHasWaitNextEvent is checked. If the System 7 Finder or MultiFinder is running, gHasWaitNextEvent is set to TRUE; otherwise, it is set to FALSE.

Although the variable used for this purpose is called gHasWaitNextEvent in Listing 5-1, it does not have any standard name; it is declared and named by the application being executed. It is also the application's responsibility to find out what kind of system is running and to set the variable's value accordingly.

In the event loop shown in Listing 5-1, if gHasWaitNextEvent is set to TRUE, then WaitNextEvent is called. If gHasWaitNextEvent is set to FALSE, the main event loop calls GetNextEvent instead.

More detailed procedures for determining what kind of system is running and for setting the gHasWaitNextEvent variable to its correct value are described later in this chapter.

▶ Using a CASE Statement in a Main Event Loop

When an application detects an event by calling WaitNextEvent or GetNextEvent, a CASE statement is generally used to determine what kind of event was detected. Then the application being executed can handle the event in an appropriate manner.

▶ The SystemTask Call

Another important feature of Listing 5-1 is the SystemTask call that precedes the GetNextEvent call. Note that a call to SystemTask appears in the IF statement that calls GetNextEvent, but not in the IF statement that calls WaitNextEvent.

SystemTask is a Desk Manager trap that handles desk accessories. GetNextEvent does not handle desk accessories by itself, so if you want to write an application that is compatible with desk accessories, you must call SystemTask before you call GetNextEvent. But desk-accessory support is built into WaitNextEvent, so you should not call SystemTask in a loop that makes a WaitNextEvent call.

Note that GetNextEvent and WaitNextEvent handle null events differently. When GetNextEvent returns a null event, it merely means that there are no events to be processed. But WaitNextEvent does not report a null event until all background processing has been completed.

▶ WaitNextEvent and the System 7 Finder

Using WaitNextEvent in an application gives the operating system permission to process background events—that is, events generated by other open applications on the desktop—when it is not being used to process events belonging to the application in the foreground. When you write a program that uses WaitNextEvent, you can tell WaitNextEvent how much processor time you need to process events for your application when it is running in the foreground, and how much time you are willing to relinquish for the processing of background events. Procedures for passing time parameters to WaitNextEvent are described later in this chapter.

▶ The Event Queue

While an event is waiting to be processed, it is stored, or posted, in a data structure known as the event queue. The event queue is a standard operating system queue. It is provided by the operating system and maintained by the Operating System Event Manager. Although you'll probably never have to access it directly, its structure is shown in Listings 5-2 and 5-3. Operating system queues are described in more detail in the Operating System Utilities chapter of *Inside Macintosh*, Volume II.

The event queue can hold a maximum of 20 events. Once an event is posted in the queue, it stays there until it is processed by WaitNextEvent or GetNextEvent, or until the queue is full. When the queue is filled to capacity, the OS Event Manager makes room for new events by clearing out older ones, starting with the oldest events and the events with the lowest priorities.

Listing 5-2. The event queue (C listing)

```
typedef struct EvQEl {
   QElemPtr   qLink            /* next queue entry */
   short      qType            /* queue type */
   short      evtQWhat         /* event code */
   long       evtQMessage      /* event message */
   long       evtQWhen         /* ticks since startup */
   Point      evtQWhere        /* mouse location */
   short      evtQModifiers;   /* modifier flags */
}  EvQEl;
```

Listing 5-3. The event queue (Pascal listing)

```
TYPE  EvQEl = RECORD
   qLink:          QElemPtr; {next queue entry}
   qType:          INTEGER;  {queue type}
   evtQWhat:       INTEGER;  {event code}
   evtQMessage:    LONGINT;  {event message}
   evtQWhen:       LONGINT;  {ticks since startup}
   evtQWhere:      Point;    {mouse location}
   evtQModifiers:  INTEGER   {modifier flags}
   END;
```

▶ The Structure of an Event Queue Record

The event queue's qLink field is a pointer to the next entry in the queue. The qType field specifies the type of the queue. In the interface file Events.p, the qType field for the event queue is defined as ORD(evType).

The other five fields in the event queue are identical to the five fields of an event record, a data structure that is defined in the interface files Events.h and Events.p. As explained later in this chapter, applications use event records to communicate with the Toolbox Event Manager.

Note ▶ **The Event Queue Header Is a Global Variable.** The header of the event queue can be found in the global variable Event Queue. For more about global variables, see Chapter 6.

▶ Syntax of GetNextEvent and WaitNextEvent

In C format, the syntax of the GetNextEvent call is:

```
pascal Boolean GetNextEvent(eventMask,theEvent)
    short eventMask;
    EventRecord *theEvent;
```

In Pascal format, the calling sequence is:

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR
    theEvent:   EventRecord) : BOOLEAN;
```

WaitNextEvent uses a syntax that is similar to that of GetNextEvent. WaitNextEvent has the same parameters as GetNextEvent, plus two others, as you'll see later in this chapter.

The eventMask and theEvent Parameters

The *eventMask* parameter in a GetNextEvent or WaitNextEvent call is a short integer that tells the Event Manager what kinds of events to report. It is provided so that a program will not have to monitor types of events in which it is not interested.

The *theEvent* argument is a pointer to a data structure called an event record. The structure of an event record is similar to the structure of the records in the OS Event Manager's event queue; it is described later in this chapter.

▶ Calling WaitNextEvent and GetNextEvent

When an application invokes the Toolbox Event Manager using a GetNextEvent or WaitNextEvent call, the Toolbox Event Manager asks the OS Event Manager for the next event to be processed. In response to this request, the OS Event Manager removes the next event to be processed from the event queue and hands it over to the Toolbox Event Manager. Then the Toolbox Event Manager reports the event to the application. Finally, the application handles the event in whatever way is appropriate.

▶ The Event Record

The Toolbox Event Manager reports events to applications by copying information from the OS Event Manager's event queue into another record called an event record. The event record, unlike the event queue, is a data structure provided by the application that calls the Toolbox Event Manager.

When you write an application that calls the Event Manager, you must place an event-record structure somewhere in your program: usually, in the global-data section of the program or in the segment of the program that contains your main event loop. The structure of an event record is modeled after the structure of the records in the OS Event Manager's event queue. In fact, the five fields of an event record are identical to the third through seventh fields of a record in the event queue.

Listing 5-4 shows the structure of an event record in C format. (To see what an event record looks like in Pascal format, refer to Listing 5-5; the fields in the record are described later in this chapter.)

Listing 5-4. Event record (C listing)

```
typedef struct EventRecord {
...short          what         /* event code */
   long           message      /* event message */
   long           when         /* ticks since startup */
   Point          where        /* mouse location */
   short          modifiers;   /* modifier flags */
}  EventRecord;
```

▶ Activate and Update Events

The Toolbox Event Manager does not spend all of its time answering calls from application programs; it also responds to direct calls from other Toolbox managers, such as the Window Manager.

For example, when a window becomes active or inactive, the Window Manager updates the windows on the screen as necessary and then informs the Event Manager that an activate event has occurred. The next time your application calls GetNextEvent or WaitNextEvent, the Event Manager informs your application that the activate event has been processed.

Similarly, when the user moves the windows on the screen around in such a way that the contents of a window must be redrawn, the Window Manager generates an update event and informs the Event Manager that an update event has occurred. The Toolbox Event Manager then reports the update event to the application being executed by copying information from the OS Event Manager's event queue into the application's event record.

When an application calls GetNextEvent or WaitNextEvent, information on the next event to be processed is copied from the OS Event Manager's event queue into the application's event record. Then the application can redraw the contents of the window. (Procedures for redrawing the contents of windows are explained in the Window Manager chapter of *Inside Macintosh.*)

Mouse Events

The Toolbox Event Manager reports two main kinds of mouse events: mouse-down events and mouse-up events. It can also recognize double-click operations. When a mouse event has occurred, the Toolbox Event Manager reports it to your application in response to a WaitNextEvent or GetNextEvent call, and you can then handle it in whatever way your program handles mouse operations.

If the user of a program moves the mouse but does not click it, an operating system event called a mouse-moved event is reported. Procedures for detecting and handling mouse-moved events are described later in this chapter.

Keyboard Events

When the user presses a key on the keyboard, the Toolbox Event Manager updates the event record to show that a keyboard event has occurred and to show what key has been pressed. Your program can then call the Text Edit routine TEKey to display the appropriate character, or to take any other kind of action you want to take in your program.

Event Management in a Nutshell

To sum up, the two Macintosh Event Managers collect events from a variety of sources, and then they report them to applications in response to the WaitNextEvent and GetNextEvent calls, one event at a time.

Kinds of Events

When a user takes an action that activates or deactivates a window—such as clicking the mouse in an inactive window and thus making it the active window—an activate event occurs. Activate events generally occur in pairs; when one window is deactivated, another is usually activated.

Activate Events

Activate events are always reported ahead of all other events. They are never placed in the event queue; in fact, the Event Manager always checks to see if there are any pending activate events before it even looks at the event queue. If there is an activate event, the Event Manager reports it immediately, and all of the events waiting in the queue just have to keep waiting.

Because of the way in which activate events are detected, there can never be more than two activate events pending at the same time. However, since they frequently occur in pairs, there often are two activate events to report: one for a window becoming inactive and a second for a window becoming active.

▶ Mouse Events

When an application receives notification of a mouse-down event, it ordinarily calls the Window Manager function FindWindow to find out where the cursor was when the mouse button was pressed. Then the application responds in whatever way is appropriate. For example, depending on the cursor location when the mouse button was pressed, an application might call

- the Menu Manager function MenuSelect
- the Desk Manager procedure SystemClick
- the Window Manager routines SelectWindow, DragWindow, GrowWindow, or TrackGoAway
- the Control Manager routines FindControl, TrackControl, or DragControl

If an application has a special way of treating a mouse click with a modifier key held down, it can determine whether or not a modifier key was down, and what key it was, by examining the modifiers field of the event record. The modifiers field is described in more detail later in this chapter.

In programs that use the Toolbox manager TextEdit to handle text editing, a double click of the mouse in a Text Edit entry field automatically selects a word; however, to respond to a double click in any other context, you must detect it yourself. You can do that by comparing the time and location of a mouse-up event with the time and location of the next mouse-down event. For more information about detecting double clicks, see the Event Manager chapter of *Inside Macintosh.*

Except for detecting double clicks and performing other specialized operations, most simple applications respond to mouse-down events but ignore mouse-up events.

Note ▶

Keeping Track of the Mouse. Until the advent of the System 6 MultiFinder, mouse movements were not reported as events; only mouse clicks were. However, you can now detect mouse movements by setting the *mouseRgn* parameter of the WaitNextEvent call, as explained later in this chapter.

▶ Keyboard Events

With a few important exceptions, the keys on both the keyboard and the numeric keypad generate key-down and key-up events when they are pressed and released. The keys that do not generate events are the Shift, Caps Lock, Command, the Control key and the Option key. They are called modifier keys.

When a key that is not a modifier key is pressed, the Macintosh generates an internal character code, and the Event Manager places the code in the event record, where it can be retrieved by the next WaitNextEvent or GetNextEvent call. The character code set used by the Macintosh is an extended version of the standard ASCII code set. The Macintosh extended character set is listed in Table 5-1.

Table 5-1. The Macintosh extended character set

Hex	*Decimal*	*Character*	*Comments*
00-02	0-2		Not used
03	3	ETX	Enter
04-07	4-7		Not used
08	8	BS	Backspace
09	9	HT	Tab
0A-0C	10-12		Not used
0D	13	CR	Return
1B	27	ESC	Escape
1C	28	FS	Left arrow
1D	29	GS	Right arrow
1E	30	RS	Up arrow
1F	31	US	Down arrow
20	32	Space	Space
21	33	!	!

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
22	34	"	"
23	35	#	#
24	36	$	$
25	37	%	%
26	38	&	&
27	39	'	'
28	40	(	(
29	41	)	0
2A	42	*	*
2B	43	+	+
2C	44	,	,
2D	45	-	-
2E	46	.	.
2F	47	/	/
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	:
3B	59	;	;
3C	60	<	<
3D	61	=	=
3E	62	>	>
3F	63	?	?
40	64	@	@
41	65	A	A
42	66	B	B

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
43	67	C	C
44	68	D	D
45	69	E	E
46	70	F	F
47	71	G	G
48	72	H	H
49	73	I	I
4A	74	J	J
4B	75	K	K
4C	76	L	L
4D	77	M	M
4E	78	N	N
4F	79	O	O
50	80	P	P
51	81	Q	Q
52	82	R	R
53	83	S	S
54	84	T	T
55	85	U	U
56	86	V	V
57	87	W	W
58	88	X	X
59	89	Y	Y
5A	90	Z	Z
5B	91	[	[
5C	92	\	\
5D	93	]	]
5E	94	^	^
5F	95	_	_
60	96	`	`
61	97	a	a
62	98	b	b
63	99	c	c

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
64	100	d	d
65	101	e	e
66	102	f	f
67	103	g	g
68	104	h	h
69	105	i	i
6A	106	j	j
6B	107	k	k
6C	108	l	l
6D	109	m	m
6E	110	n	n
6F	111	o	o
70	112	p	p
71	113	q	q
72	114	r	r
73	115	s	s
74	116	t	t
75	117	u	u
76	118	v	v
77	119	w	w
78	120	x	x
79	121	y	y
7A	122	z	z
7B	123	{	{
7C	124	|	|
7D	125	}	}
7E	126	~	~
7F	127		Not used
80	128	Ä	International keyboards
81	129	Å	Option-Shift-A
82	130	Ç	Option-Shift-C
83	131	É	International keyboards
84	132	Ñ	International keyboards

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
85	133	Ö	International keyboards
86	134	Ü	International keyboards
87	135	á	International keyboards
88	136	à	International keyboards
89	137	â	International keyboards
8A	138	ä	International keyboards
8B	139	ã	International keyboards
8C	140	å	Option-a
8D	141	ç	Option-c
8E	142	é	International keyboards
8F	143	è	International keyboards
90	144	ê	International keyboards
91	145	ë	International keyboards
92	146	í	International keyboards
93	147	ì	International keyboards
94	148	î	International keyboards
95	149	ï	International keyboards
96	150	ñ	International keyboards
97	151	ó	International keyboards
98	152	ò	International keyboards
99	153	ô	International keyboards
9A	154	ö	International keyboards
9B	155	õ	International keyboards
9C	156	ú	International keyboards
9D	157	ù	International keyboards
9E	158	û	International keyboards
9F	159	ü	International keyboards
A0	160	†	Option-t
A1	161	°	Option-Shift-8
A2	162	¢	Option-4
A3	163	¶	Option-7
A4	164	§	Option-S
A5	165	•	Option-8

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
A7	167	ß	Option-s
A8	168	®	Option-r
A9	169	©	Option-g
AA	170	™	Option-2
AB	171	´	Option-e
AC	172	¨	Option-u
AD	173	≠	Option-=
AE	174	Æ	Option-Shift-'
AF	175	Ø	Option-Shift-O
B0	176	∞	Option-5
B1	177	±	Option-+
B2	178	≤	Option-<
B3	179	≥	Option->
B4	180	¥	Option-Y
B5	181	µ	Option-M
B6	182	∂	Option-D
B7	183	∑	Option-W
B8	184	∏	Option-Shift-P
B9	185	π	Option-Shift-P
BA	186	∫	Option-B
BB	187	ª	Option-9
BC	188	º	Option-0 (zero)
BD	189	Ω	Option-Z
BE	190	æ	Option-'
BF	191	ø	Option-O
C0	192	¿	Option-Shift-?
C1	193	¡	Option-1
C2	194	¬	Option-L
C3	195	√	Option-V
C4	196	ƒ	Option-F
C5	197	≈	Option-X
C6	198	∆	Option-J
C7	199	«	Option-\

Table 5-1. The Macintosh extended character set (continued)

Hex	*Decimal*	*Character*	*Comments*
C8	200	»	Option-Shift-\
C9	201	…	Option-;
CA	202	Nonbreaking space	Option-Space
CB	203	À	International keyboards
CC	204	Á	International keyboards
CD	205		International keyboards
CE	206	Œ̃	Option-Shift-Q
CF	207	œ	Option-Q
D0	208	–	Option-Hyphen
D1	209	—	Option-Shift-Hyphen
D2	210	“	Option-[
D3	211	”	Option-Shift-[
D4	212	‘	Option-]
D5	213	’	Option-Shift-]
D6	214	÷	Option-/
D7	215	◊	Option-Shift-V
D8	216	Ÿ	International keyboards

Modifier Keys and Character Keys

Although modifier keys do not generate any key codes themselves, the use of a modifier key is always reported by the Event Manager. If the Shift, Caps Lock, or Option key is held down while a character key is pressed, the code that the character key generates is changed. If the Command key is held down while a character key is pressed, it does not change the character code that is generated, but the fact that the Command key was down is reported in the event record.

The Shift key is the only modifier key that has any effect on character codes generated by the keys on the numeric keypad.

▶ Disk Events

A disk-inserted event occurs when the user inserts a disk into a disk drive or takes any other action that requires a volume to be mounted. For example, activating a hard-disk drive that contains several volumes may cause a disk-inserted event to be reported.

▶ Auto-key Events

An auto-key event is generated when the user holds down a repeating key for a predetermined length of time. However, an auto-key event is posted only if all of these conditions are true:

- Auto-key events have not been disabled. (The disabling of events is covered later in this chapter.)
- No higher priority event is pending.
- The user is currently holding down a character key.
- A predetermined period of time has elapsed since the last key-down or auto-key event occurred.

Two different time intervals are associated with auto-key events. First, an auto-key event is generated after a key has been held down for a predetermined period of time. Then another auto-key event is generated each time a second specified period of time elapses.

The period of time that must pass before an initial auto-key event is generated is called the auto-key threshold. The time that must elapse before subsequent auto-key events are reported is called the auto-key rate. The default values are 16 ticks (sixtieths of a second) for the auto-key threshold and four ticks for the auto-key rate. You can change these values by adjusting the keyboard touch and the key-repeat rate using the Control Panel desk accessory.

The current values of the auto-key threshold and the auto-key rate are stored in the global variables KeyThresh and KeyRepThresh, respectively. For more information about global variables, refer to Chapter 7.

▶ Update Events

When text or graphics in a window must be drawn or redrawn—for example, when the user opens, closes, activates, or moves a window—an update event is generated.

Although update events are not placed in the event queue, they are not handled immediately. Instead, when higher priority events are not being processed, the Event Manager checks to see if there are windows whose contents need to be drawn or redrawn. If a window that must be drawn or redrawn is found, an update event for that window is reported.

If two or more windows need to be updated, the first update event reported is the one for the frontmost window. Update events for other windows are subsequently reported, one at a time, with the window that is frontmost always receiving the highest priority.

Null Events

When the Event Manager has no events to report, and no background task is pending, it reports a null event. In programs that were designed before the advent of MultiFinder, programs sometimes performed garbage collection and other kinds of time-consuming tasks each time a null event was received. But that is not a good idea anymore. When you run a program under System 7 or MultiFinder, null events are not reported until pending background events have been processed—and that means that there may not be enough time left to perform a time-consuming operation. So you should not wait until you receive a null event to perform a task that takes a while; instead, you should perform the task when the need for it arises.

Other Kinds of Events

Three other events that can be reported by the Event Manager are device driver events, network events, and application-defined events.

Device driver events can be generated by device drivers in certain situations; for example, a driver might be set up to report an event when a transmission of data is interrupted. More information about driver events can be found in the chapters on specific device drivers in *Inside Macintosh*.

The AppleTalk Manager can generate network events. For details on network events, see the File Manager chapter of *Inside Macintosh*.

An application can define as many as four event types of its own and can use them for any desired purpose. Application-defined events can be placed in the event queue with the Operating System Event Manager procedure PostEvent. However, application events should be used with care. See the *Inside Macintosh* chapter on the Operating System Event Manager for more details.

▶ Event Priorities

Events are not necessarily reported in the order they occurred; some are reported ahead of others because they have a higher priority. Some events—specifically, activate and update events—are not even posted in the event queue at all. Activate events are reported ahead of all other events, and update events are reported whenever the Event Manager gets around to them; that is, when it has no higher priority events to report.

Mouse and keyboard events, unlike activate and update events, are always placed in the event queue by the OS Event Manager. The Toolbox Event Manager then reports them to the application being executed on a last-in, first-out basis, in accordance with their priorities. This list shows the kinds of events the Event Manager recognizes, arranged in order of their priorities:

1. activate events
2. mouse, keyboard, and disk events
3. auto-key events
4. update events
5. null events (no event to report)

Category 2 includes most of the event types that are monitored by application programs. Within this category, events are retrieved from the queue in the order in which they were posted. The Event Manager reports an auto-key event if a key has been held down for a specified length of time (determined by a Control Panel setting).

▶ Event Records

As mentioned earlier, the Event Manager responds to the WaitNextEvent and GetNextEvent calls by updating fields in a data structure called an event record. Listing 5-5 shows the structure of an event record in Pascal format. (The structure of an event record in C format was shown in Listing 5-4.)

Listing 5-5. Event record (Pascal listing)

```
TYPE  EventRecord = RECORD
   what:            INTEGER; {event code}
   message:         LONGINT; {event message}
   when:            LONGINT; {ticks since startup}
   where:           Point;   {mouse location}
   modifiers:       INTEGER  (modifier flags}
   END;
```

What an Event Record Contains

When a WaitNextEvent or GetNextEvent call has been made and the Event Manager has updated the fields in the event record, this is what each field contains:

- The *what* field contains an event code that tells the type of event detected.
- The *when* field tells the time the event was posted, measured in ticks since system startup.
- The *where* field tells the location of the mouse cursor at the time the event was posted, expressed in global coordinates.
- The *modifiers* field contains a set of modifier flags that reveal the state of the mouse button and modifier keys at the time the event was posted.
- The *message* field contains any additional information required for a particular type of event, such as a message telling which key the user pressed or a message telling which window is being activated.

Decoding the Event Record

The time returned in the *when* field is measured in ticks since system startup. The location of the mouse returned in the *where* field is given in global coordinates, that is, in screen coordinates rather than in coordinates that correspond to a location in a window.

The Event Code

The *what* field of an event record contains an event code identifying the type of the event that has occurred. The event codes used by the event manager are defined as constants in the MPW interface files Events.h, Events.p, and Events.a. These constants, and their numeric definitions, are shown in Listing 5-6.

Listing 5-6. Event codes

```
CONST  nullEvent    =  0;    {null}
       mouseDown    =  1;    {mouse-down}
       mouseUp      =  2;    {mouse-up}
       keyDown      =  3;    {key-down}
       keyUp        =  4;    {key-up}
       autoKey      =  5;    {auto-key}
       updateEvt    =  6;    {update}
       diskEvt      =  7;    {disk-inserted}
       activateEvt  =  8;    {activate}
       networkEvt   = 10;    {network}
       driverEvt    = 11;    {device driver}
       app1Evt      = 12;    {application-defined}
       app2Evt      = 13;    {application-defined}
       app3Evt      = 14;    {application-defined}
       app4Evt      = 15;    {application-defined}
```

▶ The Event Message

The information returned in an event record's event message depends on the event type, as shown in Table 5-2.

Table 5-2. The event message

Event Type	*Event Message*
Keyboard event	Character code, key code, and ADB address field.
Activate or update event	Pointer to a window.
Disk-inserted event	Drive number in low-order word; File Manager result code in high-order word.
Mouse-down, mouse-up, or null event	Not defined.
Network event	Handle to parameter block.
Device driver event	Varies; see chapter describing driver in *Inside Macintosh*.
Application-defined event	Defined by application.

The Event Message for Keyboard Events

The structure of an event message generated by a key-down or key-up event is shown in Figure 5-1.

The field in Figure 5-1 labeled "ASCII Character" contains the ASCII character code generated by the key or key combination pressed or released by the user.

The field labeled "Virtual key code" identifies the character key that was pressed or released by the user; this value is always the same for any given character key, regardless of any modifier keys that may have been held down when it was pressed. One function of the virtual key code field is to assign key codes to the Control key and the arrow keys. The virtual key codes assigned to the Control key and arrow keys were shown in Table 5-1.

When more than one keyboard is being used, the field labeled "ADB address" shows the Apple Desktop Bus address of the keyboard that generated the event message. More information about the Apple Desktop Bus is available in the Apple Desktop Bus chapter of *Inside Macintosh,* Volume V.

The high byte of the event message for keyboard events is not defined.

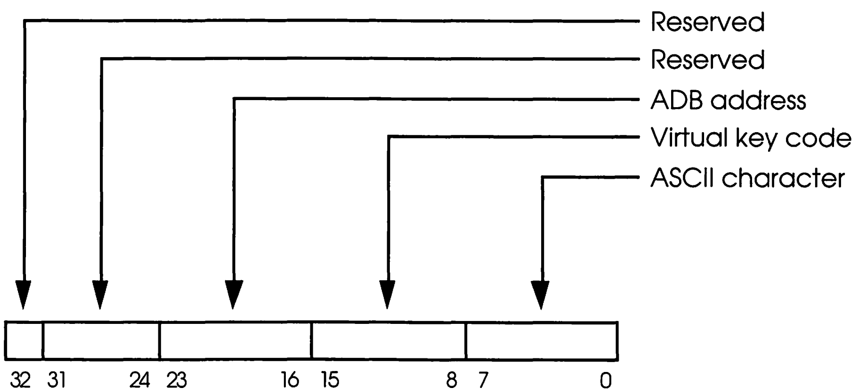


Figure 5-1. Event message for keyboard events

How Virtual Key Codes Are Generated

The virtual key codes returned by the Event Manager are derived from raw key codes generated by the Macintosh hardware. These raw key codes are translated into virtual key codes using the 'KMAP' resource in the System Folder. By modifying the 'KMAP' resource, you can change the virtual key code generated by any key on the keyboard. You can also change the ASCII codes that correspond to specific key codes by modifying the 'KCHR' resource in the System Folder.

Macintosh resources are examined in Chapter 6 of this book, and the 'KMAP' and 'KCHR' resources are described in the Resource Manager chapter of *Inside Macintosh.* More information about key mapping can be found in the Event Manager chapters in Volumes I and V of *Inside Macintosh.* Table 5-3 shows the raw key codes generated by the Control key and arrow keys on the Macintosh keyboard, and the virtual key code derived from each key code. The key codes are written in hexadecimal notation.

Table 5-3. Raw key codes and virtual key codes

Key	*Raw Key Code*	*Virtual Key Code*
Control	36	3B
Left arrow	3B	7B
Right arrow	3C	7C
Down arrow	3D	7D
Up arrow	3E	7E

The Event Message for Activate Events

The event message generated by an activate event is a pointer to the window being activated or deactivated. Additional information about the event is returned in the *modifiers* field of the event record, as described later in this chapter.

The Event Message for Update Events

The event message generated by activate and update events is a pointer to the window affected. (If the event is an activate event, additional important information about the event can be found in the *modifiers* field of the event record.)

The Event Message for Disk-Inserted Events

When a disk-inserted event is reported, the low-order word of the event message contains the drive number of the disk drive into which the disk was inserted: 1 for the Macintosh's built-in drive, and 2 for the external drive, if any. Numbers greater than 2 denote additional disk drives connected to the Macintosh. By the time an application receives a disk-inserted event, the system will already have attempted to mount the volume on the disk by calling the File Manager function MountVol; the high-order word of the event message will contain the result code returned by MountVol.

Event Messages for Other Events

When the Event Manager reports a mouse-down, mouse-up, or null event, the event message is undefined and can be ignored. The event message for a network event contains a handle to a parameter block, as described in the AppleTalk Manager chapter of *Inside Macintosh*. For device driver events, the contents of the event message depend on the situation under which the event was generated; for details, see the chapters on drivers in *Inside Macintosh*.

If you use application-defined events in a program, you can define the contents of the event message.

▶ Modifier Flags

When the Event Manager reports an activate event, the *modifiers* field of the event record tells whether the window specified in the event message is being activated or deactivated. If Bit 0 of the *modifiers* field is set, the specified window is being activated. If Bit 0 is clear, the window is being deactivated.

When a mouse or keyboard event is reported, the *modifiers* field describes the state of the modifier keys and the state of the mouse button at the time the event was posted. The flags that the *modifiers* field contains are shown in Figure 5-2.

The Events.h and Events.p interface files contain a set of predefined constants that can be used for reading the flags in the *modifiers* field. The definitions of these constants are shown in Table 5-4.

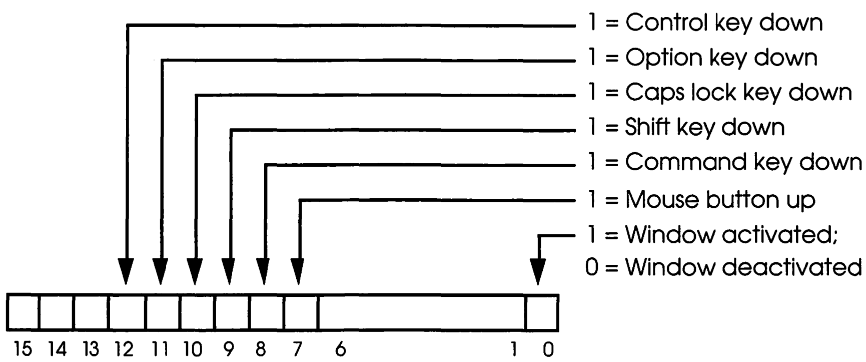


Figure 5-2. Modifier flags

Table 5-4. Modifier flags

Name of Flag	*Decimal Value*	*Hex Value*	*Meaning*
activeFlag	1	1	Set if window is being activated.
btnState	128	80	Set if mouse button is up.
cmdKey	256	100	Set if Command key is down.
shiftKey	512	200	Set if Shift key is down.
alphaLock	1024	400	Set if Caps Lock key is down.
optionKey	2048	800	Set if Option key is down.
controlKey	4096	1000	Set if Control key is down.

As previously mentioned, the activeFlag bit in the *modifiers* field provides information about activate events; it is set to 1 if the window pointed to by the event message is being activated, but cleared to 0 if the window is being deactivated.

The other bits indicate the state of the mouse button and modifier keys. Note that the btnState bit is set if the mouse button is up, whereas the bits that correspond to the four modifier keys are set if their corresponding keys are down.

▶ The Event Mask

By setting the event mask parameter of a GetNextEvent or WaitNextEvent call, you can instruct the Event Manager to report only certain kinds of events and to ignore others. For example, instead of merely requesting the next available event, you can specifically ask for the next mouse event or the next keyboard event.

Although the event mask is always available for your use, under normal circumstances it is usually best to let the Event Manager report all events to an application. Then, if there are events you do not care about, you can simply ignore them in your application. You should filter events with the event mask only when you have a good reason.

To intercept all events, you can use the event mask everyEvent, which is predefined as a constant in the Events.h and Events.p interface files. The numeric value of the everyEvent constant is -1.

The Structure of the Event Mask

An event mask is a short integer containing one bit position for each event type, as shown in Figure 5-3. Each bit in the event mask corresponds to an event code that has the same number as the position of the bit.

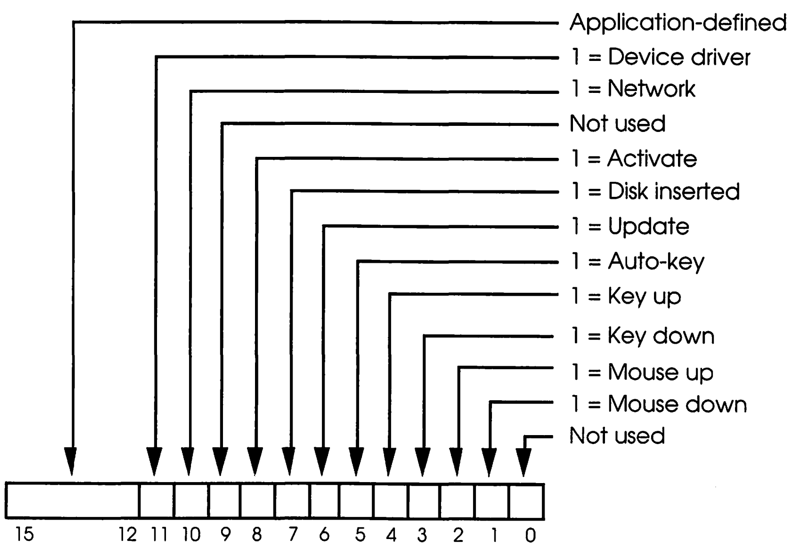


Figure 5-3. Event mask

For example, update events—which have an event of code 6—correspond to bit 6 of the event mask. So, by setting bit 6 of the event mask, you can instruct the Event Manager to report update events. Or, by clearing bit 6, you can instruct the Event Manager to ignore them.

Note that null events cannot be disabled; a null event is always reported when no other enabled events are pending, and when events generated by background applications are not being processed.

Bit 0 of the event mask is not used.

Event Mask Constants

A set of predefined constants that you can use to set the event mask is provided in the Events.h and Events.p interface files. These constants are listed in Table 5-5.

You can create any mask you need by performing addition and subtraction operations on the mask constants listed in Table 5-5. For example, to instruct the Event Manager to report key-down and auto-key events, you could set the event mask to the value

```
keyDownMask + autoKeyMask
```

To receive reports of all events except mouse events, you could use this format:

```
everyEvent - mDownMask - mUpMask
```

Table 5-5. Event mask constants

Constant	*Decimal*	*Hex*	*Meaning*
everyEvent	-1	FFFFFFFF	Don't mask any events.
mDownMask	2	2	Mask mouse-down events.
mUpMask	4	4	Mask mouse-up events.
keyDownMask	8	8	Mask key-down events.
keyUpMask	16	10	Mask key-up events.
autoKeyMask	32	20	Mask auto-key events.
updateMask	64	40	Mask update events.
diskMask	128	80	Mask disk-inserted events.
activMask	256	100	Mask activate events.
networkMask	1024	400	Mask network events.
driverMask	2048	800	Mask device driver events.
app1Mask	4096	1000	Mask application-defined events.
app2Mask	8192	2000	Mask application-defined events.
app3Mask	16384	40000	Mask application-defined events.
app4Mask	-32768	FFFF8000	Mask application-defined events.

Note ▶

The System Event Mask. The Macintosh maintains a system event mask that controls which event types the Operating System Event Manager posts in the event queue. Only event types that match the bits that are set in the system event mask are posted; all others are ignored. When the system starts up, the system event mask is set to post all events except key-up events; that is, it is initialized to

```
everyEvent - keyUpMask
```

Key-up events are not included in the system event mask because they are meaningless in most applications. However, if you write an application that uses them, you can set the system event mask to everyEvent by making the Operating System Event Manager call SetEventMask. The SetEventMask call is described in the Event Manager chapter of *Inside Macintosh.*

▶ The WaitNextEvent Call

The WaitNextEvent call, as explained earlier, is used in applications that are designed to be compatible with System 7 and with the System 6 MultiFinder. It provides applications with a method for handling background events efficiently. The syntax of the WaitNextEvent call, in C notation, is:

```
pascal Boolean
WaitNextEvent(eventMask,theEvent,sleep,mouseRgn)
    short eventMask;
    EventRecord *theEvent;
    unsigned long sleep;
    RgnHandle mouseRgn;
```

In Pascal notation, the call sequence is:

```
FUNCTION WaitNextEvent (eventMask: INTEGER; VAR
    theEvent:   EventRecord; sleep: LONGINT; mouseRgn:
    RgnHandle) :    BOOLEAN;
```

▶ Writing an Event Loop

Listing 5-7 is a fragment of code that shows how the GetNextEvent and WaitNextEvent calls can be used in an MPW Pascal program. The fragment is taken from a sample program named Creation.p, which is listed in Appendix C. The program is named Creation because it is a template that you can use to create your own System 7- and MultiFinder-aware applications.

Listing 5-7. Main event loop of the Creation.p program

```
{$S Main}

PROCEDURE EventLoop;

  VAR
    cursorRgn: RgnHandle;
    gotEvent: BOOLEAN;
    ignoreResult: BOOLEAN;
    mouse: Point;
    key: Char;
  BEGIN
    cursorRgn := NewRgn; {we'll pass an empty region
to WNE the first time thru}
    REPEAT
      IF gHasWaitNextEvent THEN
        ignoreResult := WaitNextEvent(everyEvent,
myEvent, GetSleep, cursorRgn)
      ELSE
        BEGIN
        SystemTask;
        gotEvent := GetNextEvent(everyEvent,
myEvent);
        END;
      AdjustCursor;
      CASE myEvent.what OF
        mouseDown: DoMouse;
        keyDown, autoKey:
          BEGIN
          key := CHR(BAND(myEvent.message,
charCodeMask));
          IF BAND(myEvent.modifiers, cmdKey) <> 0
```

Listing 5-7. Main event loop of the Creation.p program (continued)

```
THEN
              BEGIN { Command key down }
              IF myEvent.what = keyDown THEN
                BEGIN
                AdjustMenus;
                DoMenu(MenuKey(key));
                END; {IF}
              END
            ELSE
              DoKey;
            END; {keyDown}

          activateEvt:
DoActivate(BAND(myEvent.modifiers, activeFlag) <> 0);
          updateEvt: DoUpdate;
          nullEvent: IF (textH <> NIL) THEN
             IF (FrontWindow = myWindow) THEN
               TEIdle(textH);
          kOSEvent:
            CASE BAND(BROTL(myEvent.message, 8), $FF) OF
              kMouseMovedMessage: TEIdle(textH);
              kSuspendResumeMessage:
                BEGIN
                gInBackground := BAND(myEvent.message,
                   kResumeMask) = 0;
                DoActivate(NOT gInBackground);
                END;
            END;
        END;
      UNTIL quit;
      PrClose;
    END; {EventLoop}
```

▶ Using the gHasWaitNextEvent Variable

Before the Creation.p program calls WaitNextEvent or GetNextEvent, it checks the setting of the *gHasWaitNextEvent* variable, as explained earlier in this chapter. If *gHasWaitNextEvent* is TRUE, then the program calls WaitNextEvent. If not, it calls GetNextEvent.

In the initialization section of the Creation.p program, *gHasWaitNextEvent* is assigned a Boolean value that tells the program's main loop whether it should call WaitNextEvent or GetNextEvent in its main loop.

If the program is running on a system that has a WaitNextEvent trap—and thus supports the use of WaitNextEvent—then *gHasWaitNextEvent* is given a value of TRUE, and the program's main loop calls WaitNextEvent. If the environment in which the program is running does not support the use of WaitNextEvent, then the *gHasWaitNextEvent* variable is assigned a value of FALSE, and GetNextEvent is called instead of WaitNextEvent in the main loop of the program.

If the program decides to call GetNextEvent rather than WaitNextEvent, a SystemTask call is made before GetNextEvent is called, so that the application can handle desk accessories. A SystemTask call is not necessary if WaitNextEvent is used, as explained earlier in this chapter.

▶ Setting gHasWaitNextEvent

The *gHasWaitNextEvent* variable is declared in the data section of the Creation.p program. This is its declaration:

```
Boolean      gHasWaitNextEvent
```

After the *gHasWaitNextEvent* variable has been declared, its value is set in the initialization segment of the program.

```
gHasWaitNextEvent := TrapAvailable(_WaitNextEvent,
    ToolTrap);
```

▶ Using Gestalt

To find out whether the Macintosh being used supports the use of background processing, you should use the operating system call Gestalt. The Gestalt call enables applications to determine important information about a large number of machine-dependent features. For example, Gestalt can tell you:

- what model of the Macintosh is running the application
- the type of CPU is currently being used
- what version of the System file is currently running
- how much RAM is available
- how much virtual memory is available, if any
- the kind of keyboard that is attached to the computer
- whether a floating-point processing unit is being used, and if so, which one
- whether a memory management unit (MMU) is available, and if so, what kind
- the versions of various drivers and managers in the system
- the version of QuickDraw currently running
- whether the A/UX operating system is being used

Prior to the introduction of System Software Version 6.0.4, the operating system calls Environs and SysEnvirons were used to determine hardware and software characters of the Macintosh operating environment. With the introduction of System 6.0.4, a new operating system manager called the Gestalt Manager replaced both of those calls. That was a significant step forward, because Gestalt is simpler to use and provides more information than the Environs and SysEnvirons routines.

Also included in the Gestalt Manager are two other functions: one that enables an application to add new features to Gestalt, and another that allows an application to change the function used by Gestalt to retrieve the features of various drivers and managers.

Important ▶

SysEnvirons Still Works. For the sake of backwards compatibility, the SysEnvirons call is included in System 7.0. In System 7.0, SysEnvirons calls Gestalt, so applications that use SysEnvirons still execute correctly under System 7.0.

Nevertheless, Apple recommends that developers no longer use the Environs or SysEnvirons calls because they focus on ROM versions, not on the specific software features available in various operating environments. So many combinations of features are now available on various models of the Macintosh that it is easier and safer to find out about specific features than it is to think only in terms of ROM versions. When an application uses Gestalt, it can simply request the information it needs, and not be concerned with what kind of ROM is being used.

In the Creation.p program, the SetEnvirons call is used rather than Gestalt. As an exercise, you might try updating Creation.p so that it uses Gestalt.

▶ Gestalt Manager Calls

In System 7.0, the Gestalt Manager has three calls: Gestalt, NewGestalt, and ReplaceGestalt. Gestalt is used to obtain information about software or hardware components available on the Macintosh currently in use. NewGestalt can be used to add new software modules, such as drivers and patches, to the operating system. And ReplaceGestalt can be used to replace Toolbox and operating system procedure and functions with some other procedure or function.

Although Gestalt is a very important call, most applications do not need to use either NewGestalt or ReplaceGestalt.

▶ Selector Codes

When an application needs information about a specific software or hardware feature, it can obtain the information by passing Gestalt a selector code (sometimes referred to simply as a selector) as a parameter. The selector code tells Gestalt what kind of information the application needs.

There are two kinds of selector codes: predefined selector codes, which are always recognized by Gestalt, and application-defined selector codes, which applications may "register" with Gestalt by calling the NewGestalt function.

Predefined selector codes are divided into two categories:

- environmental selectors: codes that return information that an application can use to guide its actions
- informational selectors: codes that provide information only, and should never be used to determine whether a feature exists.

Listing 5-8 shows the environmental selectors that you can use to obtain information about the current operating environment. The selectors are shown as they are defined in the Pascal interface file used by the System 7.0 Event Manager.

Listing 5-8. Environmental selectors

```
CONST
gestaltVersion              = 'vers';  {Gestalt version}
gestaltAddressingModeAttr   = 'addr';  {addressing mode
                                        attributes}
gestaltAliasMgrAttr         = 'alis';  {Alias Mgr
                                        attributes}
gestaltAppleTalkVersion     = 'atlk';  {AppleTalk
                                        version}
gestaltAUXVersion           = 'a/ux';  {A/UX version if
                                        present}
gestaltCTBVersion           = 'ctbv';  {Comm Toolbox
                                        version}
gestaltDBAccessMgrAttr      = 'dbac';  {Database Access
                                        Mgr attrs}
gestaltEditionMgrAttr       = 'edtn';  {Edition Mgr
                                        attributes}
gestaltAppleEventsAttr      = 'evnt';  {AppleEvents
                                        attributes}
gestaltFolderMgrAttr        = 'fold';  {Folder Mgr
                                        attributes}
gestaltFontMgrAttr          = 'font';  {Font Mgr
                                        attributes}
gestaltFPUType              = 'fpu ';  {FPU type}
```

Listing 5-8. Environmental selectors (continued)

```
gestaltHardwareAttr         = 'hdwr';   {hardware
                                         attributes}
gestaltHelpMgrAttr          = 'help';   {Help Mgr
                                         attributes}
gestaltKeyboardType         = 'kbd ';   {keyboard type}
gestaltLowMemorySize        = 'lmem';   {low-memory area
                                         size}
gestaltLogicalRAMSize       = 'lram';   {logical RAM
                                         size}
gestaltMiscAttr             = 'misc';   {miscellaneous
                                         attributes}
gestaltMMUType              = 'mmu ';   {MMU type}
gestaltNotificationMgrAttr = 'nmgr';   {Notification Mgr
                                         attrs}
gestaltOSAttr               = 'os';     {O/S attributes}
gestaltLogicalPageSize      = 'pgsz';   {logical page
                                         size}
gestaltPPCToolboxAttr       = 'ppc ';   {PPC Toolbox
                                         attributes}
gestaltPowerMgrAttr         = 'powr';   {Power Mgr
                                         attributes}
gestaltProcessorType        = 'proc';   {processor type}
gestaltParityAttr           = 'prty';   {parity
                                         attributes}
gestaltQuickdrawVersion     = 'qd';     {QuickDraw
                                         version}
gestaltPhysicalRAMSize      = 'ram ';   {physical RAM
                                         size}
gestaltResourceMgrAttr      = 'rsrc';   {Resource Mgr
                                         attributes}
gestaltScriptMgrVersion     = 'scri';   {Script Mgr
                                         version}
gestaltScriptCount          = 'scr#';   {# of active
                                         script systems}
gestaltSoundAttr            = 'snd ';   {sound
                                         attributes}
gestaltTextEditVersion      = 'te';     {TextEdit
                                         version}
gestaltTimeMgrVersion       = 'tmgr';   {Time Mgr
                                         version}
gestaltVMAttr               = 'vm  ';   {virtual memory
                                         attributes}
```

Listing 5-9 shows Gestalt's informational selectors: codes that are provided for informational purposes only. Applications can display the information returned when these selectors are used, but should never use the information as an indication of what software features or hardware may be available. The selectors are shown as they are defined in the Pascal interface file used by the System 7.0 Event Manager.

Listing 5-9. Informational selectors

```
CONST
gestaltMachineType        = 'mach';  {machine type}
gestaltROMSize            = 'rom ';  {ROM size}
gestaltROMVersion         = 'romv';  {ROM version}
gestaltSystemVersion      = 'sysv';  {System file
                                     version}
```

▶ Response Parameters

If Gestalt can determine the information that an application has requested, the information is returned in a parameter known as a response parameter. If Gestalt cannot obtain the requested information, it returns an error code. Thus you should always check the result code returned by Gestalt to make sure that the response parameter contains meaningful information.

▶ Determining Whether the Gestalt Manager Is Available

The Gestalt Manager exists only in System Software Versions 6.0.4 and later (and is provided in ROM on some models of the Macintosh, such as the Macintosh IIci and Portable), so you should make certain that it is actually available before attempting to call it.

If you are using Version 3.2 or later of MPW, and you are not programming in assembly language, it is not necessary to make a specific check for the presence of the Gestalt Manager; MPW 3.2 has glue routines that allow you to call Gestalt even if it is not in ROM or in the System file of the computer being used.

However, if you are using an older version of MPW, or if you are programming in assembly language, this glue is not provided, so you must check to make sure that Gestalt is available before you call it.

Listing 5-10 shows how you can determine whether the Gestalt Manager is available.

Listing 5-10. Determining whether Gestalt is available

```
FUNCTION GestaltAvailable: Boolean;
    CONST
      _Gestalt    =   $A1AD;
    BEGIN
      GestaltAvailable := TrapAvailable(_Gestalt);
    END;
```

Calling the Gestalt Manager

Once you have determined whether the Gestalt Manager is available, you can call Gestalt to determine the hardware and software characteristics of the current Macintosh operating environment.

Listing 5-11 shows how the Gestalt call is used in the Creation.p program.

Listing 5-11. Calling the Gestalt Manager

```
IF hasGestalt THEN BEGIN
  myErr := Gestalt(gestaltTimeMgrVersion, myFeature);
  IF myErr <> noErr THEN
    DoError(myErr);
END;
```

On with the Program

Once you know how the WaitNextEvent and GetNextEvent calls work, the rest of the Create.p program's main event loop is quite straightforward. As you can see by looking again at Listing 5-7, the program calls WaitNextEvent or GetNextEvent (depending on what kind of system is running), and then handles each event reported by the Toolbox Event Manager.

Before the program starts processing events, however, it calls a procedure named AdjustCursor. AdjustCursor, as you'll see when the complete program is presented in Chapter 8, adjusts the mouse cursor depending on where it is on the screen. If it is in the content region of a window, it is displayed as a text-style I-bar cursor. If it moves outside a window, it becomes a pointer.

After the AdjustCursor procedure is called, the program starts processing events by calling routines that handle them. Since the program is a text-style application, it makes the call TEIdle when it receives a null event. TEIdle, as explained in the Text Edit chapter of *Inside Macintosh*, causes the Text Edit insertion point to flash when the mouse cursor is inside a TextEdit record. In the Creation.p program, the entire content region of the active window is a TextEdit record, so the insertion point flashes if it is inside the program's window.

If the Event Manager detects a Type 4 application event—that is, a suspend or resume event, or a mouse-moved event—it reports the event using the constant kOSEvent. The kOSEvent constant is defined as a Type 4 application event in the header section of the Creation.p program, as shown in Listing 5-12. If Creation.p receives notification that a kOSEvent has occurred, it uses the algorithms shown in Listing 5-7 to determine whether the event was a suspend event, a resume event, or a mouse-moved event, and then responds appropriately.

Listing 5-12. Constants defined in Creation.p

```
kOSEvent                = app4Evt;
kSuspendResumeMessage   = 1;
kResumeMask             = 1;
kMouseMovedMessage      = $FA;
```

By using the other two constants defined in its header section, Creation.p sets kMouseMovedMessage to TRUE if the reported event is a mouse-moved event, and sets kSuspendResumeMessage to TRUE if the event is a suspend event or a resume event. If the event is a mouse-moved event, Creation.p simply calls TEIdle.

If the event is a suspend event or a resume event, the program checks a global variable called *gInBackground* to see whether it is running in the foreground or the background. Then it calls a procedure named DoActivate, which activates the application's window if the program is moving from the background into the foreground, but it deactivates the window if the program is moving from the foreground into the background.

The DoActivate procedure and the other event-processing routines called in Listing 5-7 are described in Chapter 8.

▶ The sleep Parameter

The *eventMask* and *theEvent* parameters in a WaitNextEvent call are the same as the corresponding parameters in a GetNextEvent call. In the *sleep* parameter, you can specify how much time you want to relinquish in your main event loop for the processing of background events: that is, for events processed by other applications on the desktop. When a background event is not being processed, the Toolbox Event Manager reports a null event to your application.

The value in the *sleep* parameter is specified in ticks, or sixtieths of a second. If you specify a *sleep* parameter of 0, the Event Manager gives your application as much processor time as possible, and background events are still allocated a minimal amount of time to process events. A *sleep* parameter of 60 allocates only one null event per second to your application.

If you want to calculate a sleep period instead of using a constant—for example, if you want the value of *sleep* to be the length of time that it takes the text cursor to flash once during a program that uses TextEdit—you can write an algorithm to establish a sleep period and call a routine that performs your algorithm to fill in the *sleep* parameter.

That is the technique that is used to set the WaitNextEvent *sleep* parameter in the event loop shown in Listing 5-7. The WaitNextEvent call in Listing 5-7 is written like this:

```
ignoreResult := WaitNextEvent
    (everyEvent,myEvent,GetSleep,cursorRgn)
```

The *sleep* parameter in this example is a call to a function named GetSleep. The GetSleep function, which appears later in the program, is shown in Listing 5-13.

Listing 5-13. The GetSleep function

```
{$S Main}
FUNCTION GetSleep: LONGINT;
VAR
    sleep:  LONGINT;
    window: WindowPtr;

BEGIN
    sleep := MAXLONGINT;{default value for sleep}
IF NOT gInBackground THEN BEGIN
        window := FrontWindow;
        IF IsAppWindow(window) THEN BEGIN
            WITH textH^^ DO
                IF selStart = selEnd THE
                    sleep := GetCaretTime;
        END;
    END;
    GetSleep := sleep;
END; {GetSleep}
```

The GetSleep function decides what it should do by checking the *gInBackground* variable to see if Creation.p is running in the background, and then checking to see if the window that is currently the front window belongs to Creation.p. If both of those conditions are true, the program calls the Toolbox Event Manager routine GetCaretTime, which tells how long it takes (in ticks, or sixtieths of a second) to flash the text cursor's caret—the bar that marks the insertion point in editable text. Creation.p uses the value returned by GetCaretTime to set the WaitNextEvent call's *sleep* parameter.

▶ The mouseRgn Parameter

The *mouseRgn* parameter of the WaitNextEvent call is a handle to a region. Its data type, RgnHandle, is defined in the MPW interface files QuickDraw.h and QuickDraw.p.

If you specify a *mouseRgn* parameter in a WaitNextEvent call, the Event Manager reports a mouse-moved event each time the mouse cursor strays outside the specified region. If you place a zero in the *mouseRgn* parameter, mouse-moved events are not generated.

You can use the *mouseRgn* parameter of the WaitNextEvent call as a convenient means of determining when the shape of the mouse cursor should be changed. When the cursor moves outside the region you have defined as *mouseRgn*, you can change the shape of the cursor and then change the value of *mouseRgn* to reflect the cursor's new position. If the cursor moves back into the region it previously left, the Event Manager will report another mouse-moved event. You can then change the cursor back to its original shape and reset *mouseRgn* back to its original value.

▶ How the Event Manager Works

Now that you have seen how an event loop works, we are ready to take a closer look at how applications can process the various kinds of events reported by the Event Manager.

▶ The EventAvail Call

In some cases, a program may just want to look at a pending event and leave it available for subsequent retrieval by GetNextEvent or WaitNextEvent. In that case, the program can make the Event Manager call EventAvail instead of calling GetNextEvent or WaitNextEvent.

EventAvail reports the event by updating the event record, but it does not remove the event from the event queue. Thus an application can call EventAvail and inspect the updated contents of the event record, but leave the event unprocessed so that it can be retrieved again by a GetNextEvent or WaitNextEvent call. Procedures for using the EventAvail call are explained in the Event Manager chapter of *Inside Macintosh.*

Each time a program calls GetNextEvent, the Toolbox Event Manager retrieves the next event that should be processed from the OS Event Manager, and then tests the call against the event mask provided by the application being executed. If the event is one that the application is interested in, GetNextEvent returns a Boolean value of TRUE. If the application has no interest in the event, GetNextEvent returns a Boolean value of FALSE.

▶ Handling Keyboard Events

When an application receives notification of a key-down event from the Toolbox Event Manager, the first thing it usually does is check the *modifiers* field to see whether the key was pressed with the Command key down. If it was, the user may have selected a menu item. To determine whether this was the case, you should pass the character that the user typed to the Toolbox call MenuKey, as explained in Chapter 8 of this book and the Menu Manager chapter of *Inside Macintosh,* Volume I. If MenuKey reports that the key-down event was not menu-related, you can treat the event as a normal key-down event.

▶ Handling Activate Events

The Window Manager handles much of the housekeeping associated with activate events, such as highlighting and unhighlighting windows. But applications must take certain other actions, such as drawing or hiding scroll bars, and highlighting or unhighlighting text displayed in a window. For more information on what to do with a window when you receive notification of an activate event, see the Window Manager chapter of *Inside Macintosh,* Volume I.

▶ Handling Update Events

When an application that you have written receives an update event for a window that it owns, you are responsible for updating the window. The usual procedure for updating a window is to make the Window

Manager call BeginUpdate, draw the window's contents, and then make the Window Manager call EndUpdate.

The procedure for updating windows varies from program to program; for example, updating a window in which only TextEdit text is displayed is very different from updating a window containing graphics drawn with QuickDraw calls. Detailed procedures for handling update events can be found in the Window Manager and QuickDraw chapters of *Inside Macintosh*.

▶ Handling Disk-Inserted Events

The easiest way to handle disk-inserted events is to use the Standard File Package, which can handle disk-inserted events for you during standard-style opening and saving operations. If you want to respond to disk-inserted events at other times—for example, to an insertion of a disk when standard file operations are not taking place—you must write code to handle such operations.

Before an application receives a disk-inserted event, the system attempts to mount the disk by calling the File Manager function MountVol. So the application should examine the result code returned by the File Manager in the high-order word of the event message. If the result code indicates that the attempt to mount the volume was unsuccessful, the application might take some appropriate action, such as calling the Disk Initialization Package function DIBadMount. More details on using the File Manager and Disk Initialization Package functions are available in the chapters on those managers in *Inside Macintosh*.

▶ Other Event Manager Calls

Other Toolbox Event Manager calls that you may make from time to time are:

- GetMouse, which returns to current location of the mouse.
- Button, which returns TRUE if the mouse button is held down and FALSE if it is not.
- StillDown, which returns TRUE if the mouse button is still down as a result of a previous mouse-down event.
- WaitMouseUp, which works like StillDown but removes the preceding mouse-up event before returning FALSE if the button is not still down from the previous press of the button.

- TickCount, which returns the current number of ticks (in sixtieths of a second) since the system started up.
- GetDblTime, which shows how much time must elapse (in ticks) between a mouse-up event and a mouse-down event for the two events to be considered a double click.
- GetCaretTime, which returns the time (in ticks) between blinks of the caret, or the bar that marks the insertion point in editable text.

For more details on these calls, refer to the Event Manager chapter of *Inside Macintosh.*

▶ The OS Event Manager

The Operating System Event Manager detects low-level, hardware-related events: mouse, keyboard, disk-inserted, device driver, and network events. It stores information about these events in the event queue so that they can then be reported by the Toolbox Event Manager.

The OS Event Manager also provides other managers, such as the Window Manager, with low-level operating system routines that access the queue. These calls are similar to the Toolbox Event Manager's GetNextEvent and EventAvail calls. The OS Event Manager also reports activate, update, and Type 4 application events, which are not kept in the event queue.

In addition, the OS Event Manager has functions and procedures that application programs can use to post their own events into the event queue.

▶ System 7 and the Event Manager

Until the advent of System Software Version 7, the Event Manager recognized two main kinds of events:

- events that report actions by the user (such as pressing the mouse button, typing on the keyboard, or inserting a disk)
- events that report occurrences arising from sources other than the user (such as events generated by device drivers).

(There is also a third type of event, a null event, which the Event Manager returns if there are no other events to report. Null events are supported by all Macintosh systems, including System 7.)

New Events in System 7

In System Software Version 7.0, the Event Manager recognizes three main kind of events:

- low-level events, traditional user-initiated events, such as mouse and keyboard events
- operating system events, which inform applications of changes in their operating status
- high-level events, which allow applications to communicate with one another by putting events in the event queue of the receiving application

High-level events are reported to an application using a new event type defined by a new constant: kHighLevelEvent. The kHighLevelEvent constant is defined this way in the System 7 Event Manager interface file:

```
CONST  kHighLevelEvent  = 23;
```

In an application designed to be used with System 7, you can call high-level events by simply adding the kHighLevelEvent constant to your main event loop, in this fashion:

```
PROCEDURE DoEvent(event: EventRecord);
BEGIN
  CASE event.what OF
    nullEvent:
      DoIdle;
    mouseDown:
      DoMouseDown(event);
    mouseUp:
      DoMouseUp(event);
    keyDown, autoKey:
      DoKeyDown(event);
    activateEvt:
      DoActivate(event);
    updateEvt:
      DoUpdate(event);
    kOSEvent:
      DoOSEvent(event);
    kHighLevelEvent:
      DoHighLevelEvent(event);
  END;
END; {DoEvent}
```

By the Way ▶

The kOSEvent Type. Under System 7, operating system events are of type kOSEvent and are assigned the event code previously assigned to app4Evts (Type 4 application events). In the System 7 Event Manager's interface files, kOSEvent is a constant defined this way:

```
CONST  kOSEvent   = 15;
```

Once you have placed a kHighLevelEvent constant in your application's main event loop, you can write routines to handle high-level events and place them elsewhere in your program.

In order to manage communication with other applications, your application must define the set of high-level events it responds to and let other applications know what kinds of events it accepts. Other programs can then interact with your application.

▶ AppleEvents

System Software Version 7.0 introduces a special subcategory of high-level events, called AppleEvents. AppleEvents are events that are common to almost all applications designed to be used with System 7.

You can use AppleEvents to communicate with applications running on the same computer or on other computers.

Some AppleEvents are required in any application that supports any AppleEvents; these are known as required AppleEvents. The required AppleEvents in System 7 are

- Open Application, which opens applications
- Open Documents, which opens specified documents
- Print, which prints specified documents
- Quit, which terminates and exits an application
- Setup, which updates an application's menu items
- Get, which returns values of specified properties

Every AppleEvent belongs to one of three categories:

1. **Standard AppleEvents:** AppleEvents to which most or all applications respond. These events are defined by Apple.
2. **Registered AppleEvents:** AppleEvents defined by an application developer, a group of developers, or an interest group and regis-

tered with Macintosh Developer Technical Support. Registered AppleEvents can be defined for a single application, a suite of applications from a single developer, or applications in the same functional area, such as graphics or word processing. By registering an AppleEvent, you can make the event and its definitions public.

3. **Unregistered AppleEvents:** AppleEvents that application developers choose not to make public. Unregistered AppleEvents are used for private communication between applications.

▶ High-Level Events and the Event Record

As explained earlier in this chapter, the event record filled in by the Toolbox call WaitNextEvent has this structure:

```
TYPE EventRecord =
RECORD
  what:       Integer;  {event code}
  message:    LongInt;  {event message}
  when:       LongInt;  {ticks since startup}
  where:      LongInt;  {mouse location}
  modifiers:  Integer   {modifier flags}
END;
```

When an application receives a high-level event, the *what* field of the event record contains the event code defined by kHighLevelEvent, and the *when* field contains the number of ticks since the system last started up when the event is posted.

For high-level events, two fields of the event record have special meanings. The *message* field and the *where* field of the event record define the specific type of high-level event reported. And the message field contains the message class of the high-level event. For example, the message field for an AppleEvent contains the value 'aevt', and the message field for an Edition Manager event contains the value 'sect'.

You can define special classes of events that are specific to your application. If you have registered your application signature with Apple, then you can use your signature to define the class of events that belong to your application.

The structure and interpretation of AppleEvents are determined by a standard protocol known as the *AppleEvent protocol,* which is defined by Apple. To ensure compatibility with other Macintosh applications, you should use the AppleEvent protocol for high-level events if possible.

All Macintosh system software that sends or receives high-level events uses the AppleEvent protocol.

Under the AppleEvent protocol, AppleEvents are used in three ways:

1. To send a request to another application. A request is an AppleEvent sent by one application that requests a service from another. For example, an application can ask another application to open a document by sending an Open Documents AppleEvent ('aevt' 'odoc') with a parameter that specifies the document to be opened. One kind of request, known as a query, requests information that is kept by an application. An example of a query is the Get AppleEvent ('aevt' 'getp') which asks an application to return the value of a specific property.
2. To send a response to an AppleEvent. For example, if one application sends a query to another, the receiving application returns the requested information by sending back an AppleEvent.
3. To notify an application of an occurrence. For example, the Edition Manager sends AppleEvents to notify subscribers that a publisher (a section of a document containing shared information) has changed.

To use AppleEvents in an application, you must also inform the operating system that your application is able to receive and process AppleEvents. To accomplish this, you need to modify your application's 'SIZE' resource, as explained in Chapter 7.

For more information about the special kinds of events used in System 7-friendly programs, see Volume VI of *Inside Macintosh*.

▶ Defining Your Own Events

Programs that use application-defined events must make the OS Event Manager call PostEvent to post the events into the event queue. You can also use PostEvent to repost events that you have removed from the event queue with GetNextEvent. For more information on defining and posting your own events, see the OS Event Manager chapter in *Inside Macintosh*, Volume II.

Another OS Event Manager call, FlushEvents, can be used to get rid of events or types of events that you do not want, or no longer want, in the event queue. You can also use FlushEvents is to get rid of any stray events left over from before your application started up.

In fact, before you start your main event loop, it is usually a good idea to call FlushEvents (with an eventMask parameter of everyEvent and a stopMask value of zero) to empty the event queue of any stray events that may have been hanging around from before your application started up—for example, keyboard events caused by keystrokes typed to the Finder.

You will probably never have any need for the other Operating System Event Manager routines: GetOSEvent, which gets an event from the event queue, removing it from the queue in the process; OSEventAvail, for looking at an event without dequeueing it; and SetEventMask, which changes the setting of the system event mask.

All OS Event Manager calls are described in the Operating System Event Manager chapter of *Inside Macintosh.*

▶ Conclusion

This chapter explained how the Toolbox Event Manager and the Operating System Event Manager are used in Macintosh Programs written under MPW. Particular emphasis was given to writing event-driven programs that work properly under System 7 and the System 6 MultiFinder.

6 ▸ MPW and the Resource Manager

There's one feature of the Macintosh that's completely invisible to casual users, but is very important to programmers. That feature is the *resource*: a block of data that can be stored in a file, shared by various applications, and read into memory any time it is needed. Resources are mysterious entities if you don't understand them—but once you do, they can be very useful ingredients of a Macintosh program.

Almost any kind of data used in a program can be stored as a resource. When you write an application, the object code of your program is stored on disk and in memory as a resource. Menus, dialogs, pictures, and icons are also stored as resources. Desktop icons are resources, and program designers often create resources of their own.

To understand resources, it helps to know that every Macintosh file is divided into two pieces called *forks*. Any data that the file contains—for example, a document in a document file or a picture in a graphics file—can be stored in a what is known as a *data fork*. Everything else in the file is stored in the *resource fork*. Although both forks are present in every file, either fork can be empty. For example, a text file might have an empty resource fork, and an application file might have an empty data fork. Figure 6-1 shows how every Macintosh file is divided into a data fork and a resource fork.

When you need access to information in a data fork, you can get it from the File Manager. Resources are handled by a different Toolbox manager known, logically enough, as the Resource Manager. The Resource Manager keeps track of all the code and data stored in a file's resource fork and provides routines that allow applications and other parts of the Toolbox to access resources.

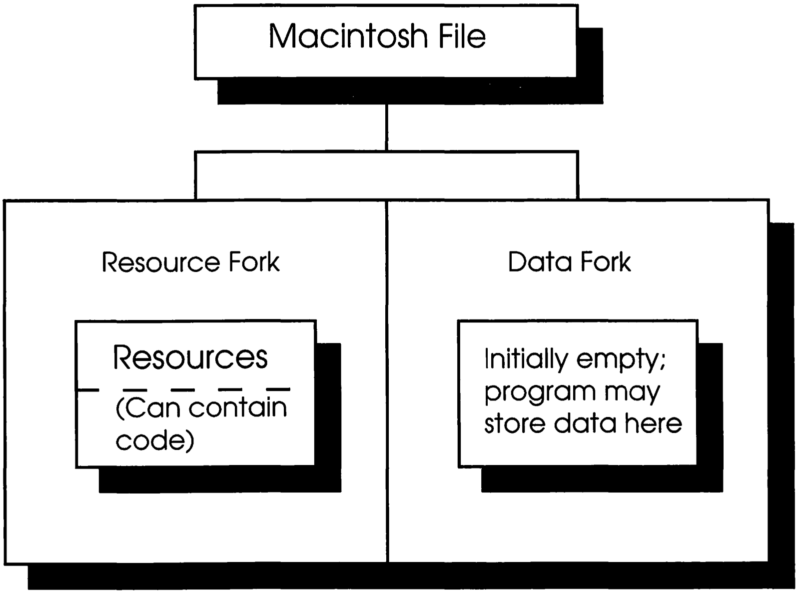


Figure 6-1. Structure of a Macintosh file

▶ Why Use Resources?

No law says an application must use resources. If you didn't want to use resources in a program, you could hard-code structures such as menus, dialogs, and icons into the program's data fork, and simply not use the Resource Manager. But, if you did that, you'd be missing out on a number of benefits that resources offer, for example,

- If you need to write programs that can be used by speakers of other languages, you can simplify the task of internationalizing your applications by storing strings, text, and structures such as menus and dialogs as resources. Then the job of translating your application from one country's language to another can be performed by a nonprogrammer. In fact, resources were originally designed as a means of facilitating the conversion of Macintosh programs into programs that could be used by speakers of foreign languages.
- When you store a structure as a resource, you do not have to keep it in memory when you aren't using it. When an application needs to use a resource, the Memory Manager automatically reads the resource into memory. When the program is finished with the

resource, and the memory that it occupied is needed for another purpose, the Memory Manager can purge the resource from memory until it is needed again.

- When you use resources in an application, you can put the source code that creates all the program's resources in the same place: in a special source file called a resource file. So, when you want to edit the source code for any resource, you always know where to find it; it isn't mixed in with the rest of the source code in the program.
- A graphics-based resource editor called ResEdit is available as a separate product. With ResEdit, you can create and edit resources interactively using the mouse, ResEdit dialogs, and screen graphics. When you have created a resource using ResEdit, you can convert it into source code using an MPW tool called DeRez. Conversely, you can create the source code for a resource file using MPW, and then compile it into resource code data with another MPW tool called Rez. With the help of these three tools—ResEdit, Rez, and DeRez—you can edit any resource either interactively (using ResEdit) or textually (using the MPW Editor).
- You can edit a resource without having to recompile the program that uses the resource. You can compile just the resource, and that takes less time.

In this chapter, we'll take a closeup look at how resources are used in Macintosh programs, particularly programs written under MPW.

▶ How Macintosh Files Are Constructed

Macintosh programmers often speak of resource files, but it's more accurate to refer to the part of a file that contains resources as a resource fork. Every Macintosh file has two forks—a resource fork and a data fork—as was shown in Figure 6-1. As you would expect, a resource fork is the part of a file in which resources are stored.

Although resources are usually thought of as belonging to applications, the truth is that applications are not the only kinds of files that can have resources. Any Macintosh file—even a document file—can have resources stored in its resource fork. For example, a document file could have a resource fork containing a special font, and a file created by a graphics program could have a resource fork containing pictures and images.

However, resources such as icons, dialogs, and fonts are not the only ingredients of an application's resource fork. The resource fork of an application file contains not only the resources used by the application, but also the object code of the application itself. Furthermore, an application's object code may be divided into segments, and if it is, the Macintosh system treats each segment as an individual resource. When an application's object code has been divided into segments—that is, separate resources—various parts of the program can be loaded and purged dynamically, which conserves memory. More information on the segmentation of programs is presented in Chapter 7.

Note ▶

'CODE' Resources in a Nutshell. When you compile and link an application, MPW assigns the application's object code a resource type of 'CODE' and stores the whole program as a resource in the application's resource fork. Normally, you'll never need to know much more than that about 'CODE' resources. Under ordinary circumstances, you'll never have to access a 'CODE' resource directly; 'CODE' resources are managed automatically by the Memory Manager and the Resource Manager.

▶ The System Resource File

In addition to the resource fork that is built into every Macintosh file, there is a system resource file that contains standard resources shared by all applications. Resources stored in the system file include the pixel images used to create the system's cursor, the bit maps of the system fonts, and various kinds of drivers. The system resource fork also contains patches to Toolbox and operating system routines.

▶ Creating and Compiling Resources

When you write the source code for a program using MPW, you must place the application's resources in a resource description file, a special kind of source-code file that is compiled separately from the rest of a program. Resources are not compiled by the MPW assembler, or by the MPW C or Pascal compiler. They are compiled by a special resource compiler, called Rez, that is bundled with the MPW development system.

To use the Rez compiler, you must provide it with a source code file written in a unique language—a language that only the Rez compiler understands. Once you have written a resource description file using the Rez language, you can invoke the MPW resource compiler by issuing the MPW command Rez, which is examined in more detail later in this chapter. The Rez compiler converts your resource file into object code, and you can then link your resources to the rest of your program by issuing the MPW command Link. More information on the Link command is provided in Chapter 8.

▶ The Rez Language

The MPW resource description language, known less formally as the Rez language, is an extensible language that looks a lot like C, but it really isn't C. It has only seven keywords that are used as commands, and they are completely different from any of the C commands. Furthermore, the purpose of the Rez language is not to execute programs, but merely to define and describe resources in a form that the Rez compiler can understand.

Note ▶

Rez! You're So Insensitive! Although the Rez language looks like a cousin to C at first glance, one noteworthy difference between the two languages is that the Rez language is case insensitive, except within delimited strings. C, as practically everybody knows, is very, very case-sensitive. In this respect, the Rez language is more like Pascal than like C.

A resource file written in the Rez language usually has a name that ends with the extension .r. When you write a source file with the .r extension, CreateMake and CreateBuildCommands recognize it as a resource description file and can compile and link it properly when you issue the Rez and Link commands.

For example, if you wrote a Pascal program called Creation.p—which is, incidentally, the name of the sample Pascal program listed in Appendix C—you would ordinarily name the program's resource code segment Creation.r, listed in Appendix D. If you wrote a C program called Creation.c, you could still call its resource file Creation.r. In fact, once a resource description file is compiled into a resource fork, it can be linked with a program written in any language. This illustrates one of the advantages of using resources in a Macintosh application: Once you have written a resource file that can be compiled by the Rez compiler, you can use it with an MPW program written in any language.

Preprocessor Directives

As you can see by looking at the Creation.r file in Appendix D, a resource definition file typically has two parts: a set of preprocessor directives and a set of statements that are used to build resources. The preprocessor directives in a resource description file begin with the symbol #, just like preprocessor directives used in C. They also have the same meanings in the Rez language that they have in C.

In a resource description file, you can use preprocessor directives to include C-style header files (files with the file name extension .h), as well as other resource description files (files with the file name extension .r) in the compilation of the file you are writing. For example, you could use these directives

```
#include "SysTypes.r"
#include "Types.r"
#include "Creation.h"
#include "menu.h"
```

to include the files SysTypes.r, Types.r, Creation.h and menu.h files in a resource description file. In this example, the Creation.h and menu.h files are application files. The SysTypes.r and Types.r files are interface files provided in MPW. The SysTypes.r file contains definitions of system variables, and the Types.r file contains definitions for most of the predefined resource types that are used in applications.

There is also an MPW interface file named Pict.r, which contains type definitions for PICT resource and for debugging PICTs, and an interface file named Cmdo.r, which contains type definitions for Commando resources.

The preprocessor directives recognized by the Rez compiler are listed in Table 6-1.

Table 6-1. Preprocessor directives used in resource description files

Directive	*Example*	*Meaning*
#include	#include *fileName*	Include source file *fileName* in compilation.
#define	#define *symbol value*	Define constant *symbol* as *value.*
#undef	#undef *symbol*	Delete definition of constant *symbol.*
#if	#if *expression*	Include text that follows in compilation if *expression* is true.

Table 6-1. Preprocessor directives used in resource description files (continued)

Directive	*Example*	*Meaning*
#else	#else	Include text that follows in compilation if preceding #if clause is not true.
#endif	#endif	Terminate #if, #ifdef, or #ifndef construct.
#ifdef	#ifdef *symbol*	Include text that follows in compilation if constant *symbol* has been defined.
#ifndef	#ifndef *symbol*	Include text that follows in compilation if constant *symbol* has not been defined.
#elif	#elif *expression*	"Else if"; include text that follows in compilation if preceding #if clause is not true, and if *expression* is true.

▶ Special Characters

In the data portion of a resource definition file, curly brackets are used to group blocks of data together, and the delimiters /* and */ are used to enclose comments. Each resource description in a resource file also includes a heading enclosed in parentheses. This heading always includes a string specifying the type of resource being described and the resource's ID number. Some headings also include a 16-bit number that specifies certain attributes that the resource has. (Headings of resource descriptions are described in more detail later in this chapter.)

For example, this code entry

```
resource ('STR ', 128) {/* Low-memory warning */
    "Warning: Memory is running low."
};
```

describes a string used as a resource. The parentheses delimit the resource's type ('STR ') and ID number (128), and the curly brackets delimit the string's contents. Resource types and ID numbers are described later in this chapter. A comment, "Low-memory warning," is enclosed in the delimiters /* and */.

Rez also supports the C++ style of comment prefix, that is, //.

Special characters used in the Rez language are listed in Table 6-2. The most important special characters are described in more detail in later sections of this chapter.

Table 6-2. Special characters used in the Rez language

Delimiters

Character	*Meaning*
( )	Delimits expressions.
{ }	Delimits blocks of data.
( )	Delimits data in headings.
/* */	Delimits comments.
'	Delimits string types.
"	Delimits strings.

Separators

Character	*Meaning*
,	Separates items in a statement.
;	Marks the end of a statement or the end of a resource description.

Operators

Character	*Meaning*
( )	Delimits expressions.
-	Unary negation operator.
!	Unary logical NOT.
~	Unary bitwise NOT.
*	Unary multiplication operator.
/	Integer division operator.
%	Modula operator (integer division remainder).
+	Addition operator.
-	Subtraction operator.
<<	Bitwise shift left.
>>	Bitwise shift right.
<	Less than.

Table 6-2. Special characters used in the Rez language (continued)

Operators (continued)

Character	*Meaning*
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
==	Equal to.
!=	Not equal to.
&	Bitwise AND.
^	Bitwise exclusive OR.
|	Bitwise OR.
&&	Logical AND.
||	Logical OR.

Miscellaneous

Character	*Meaning*
$$	Identifies a built-in Rez function. Example: $$Date.
$	Precedes hexadecimal numbers. Number that follows $ symbol is enclosed in double quotes. Example: $"1FD4".
0x	Precedes hexadecimal numbers. Number is not enclosed in quotes. Example: 0x1FD4.
#	Precedes preprocessor directives. Example: #include.
\	Escape character.
:	Specifies a range of resource ID numbers. Example: (0:32) = (resource IDs) 0 through 32.

▶ The Escape Character \

In the Rez language, the backslash character (\) is used as an escape character. You can use it to include nonprinting characters—such as return and tab characters—into a resource description. It can also be used to specify some printing characters: a quotation mark, an apostrophe, and the backslash character itself.

The backslash character is recognized as an escape character when it is enclosed in double quotation marks and is followed by a number. It is also recognized as an escape character when it precedes certain letters, as was shown in Table 6-2.

When the backslash escape character precedes a number, the number can be decimal, hexadecimal, octal, or binary. However, the number must also be preceded by a special character or character sequence that specifies its base, and it must also have a specific number of digits.

For example, when a character is expressed as an octal number, the octal number must be exactly three digits long. The Rez compiler does not do any error checking on numbers used in this fashion, to it is up to you to make the number the proper length. For a guide, see Table 6-3.

Table 6-3. The escape character \

The Escape Character Used with Numbers

Sequence	*Meaning*
"\0xF8"	Hexadecimal number (2 digits)
"\$A9"	Hexadecimal number (2 digits)
"\0d032"	Decimal number (3 digits)
"\060"	Octal number (3 digits)
"\01000110"	Binary number (8 digits)

The Escape Character Used with Letters

Sequence	*Numeric equivalent*	*Meaning*
"\b"	"\0x08"	Backspace
"\r"	"\0x0D"	Return
"\t"	"\0x09"	Tab
"\f"	"\0x0C"	Form feed
"\n"	"\0x0D"	Newline
"\v"	"\0x0B"	Vertical tab
"\""	"\0x22"	Quotation mark
"\'"	"\0x27"	Apostrophe
"\\"	"\0x5C"	Backslash
"\?"	"\0x7F"	Delete

The Resource Description Language

Each statement in a resource description file begins with a keyword that is used as a command. The seven such keywords in the Rez language are listed in Table 6-4.

Table 6-4. Keywords used in the Rez language

Keyword	*Description*
type	Type declaration: Declares a resource type description for use in a subsequent resource statement.
resource	Resource description: Specifies data for a resource type declared in a previous type statement.
data	Specifies raw data to be used as a resource.
include	Includes compiled resources from another file's data fork in the resource file being written.
read	Reads the data fork of a compiled file and includes it as a resource.
change	Changes the type, ID, name, or attributes of an existing resource.
delete	Deletes an existing resource.

The type Statement

In the resource description language (sometimes called the Rez language), the type statement is used to define the format of a resource type. Its syntax is:

```
type resourceType (idRange) {
typeSpecification....
};
```

The *resourceType* parameter in a type statement is a four-letter string enclosed in single quotation marks. The *id* parameter is a 16-bit integer. The *idRange* parameter is optional; it causes the declaration to apply only to a given resource ID or range of IDs. The *typeSpecification* parameter is described later in this chapter.

Type definitions of many predefined resources are included in the MPW interface files Types.r, SysTypes.r, MPWTypes.r, Pict.r, and

Cmdo.r, as mentioned earlier in this chapter. But you can also use type statements to define your own resource types. For example, you could use this statement to define a rectangle as a resource:

```
type 'RECT' {
    rect;
};
```

After defining a rectangle resource in this fashion, you could describe a rectangle resource in a resource description file by using the type definition 'RECT' with the Rez statement resource, as explained in the next section.

In the preceding example, note that the word "rect" is used to define the structure of the resource being defined. In MPW's resource description language, "rect" is one of a number of words that can be used in the data sections of resource definitions to specify data types. These words are called type specifications. Each of the type specifications in Table 6-5 can be used in a resource definition in the same way the specification "rect" is used in the above example.

Any number of type specifications can appear in a resource description, and they can appear in any order. Furthermore, you can use symbolic names and constants to assign values to type specifications in resource definitions, as explained later in this section. Type specifications can thus be used to create descriptions of extremely complex resources. Table 6-5 shows the formats in which type specifications are written, along with their meanings.

Table 6-5. Rez type specifications

Specification	*Description*
bitstring[*n*]	A bitstring of length *n* (maximum 32 bits).
byte	A byte (8-bit) field. Same as bitstring[8].
integer	An integer (16-bit) field. Same as bitstring[16].
longint	A long integer (32-bit) field. Same as bitstring[32].
boolean	A single bit with two possible states: true (1) and false (0). The type specification boolean defines a field one bit long. This is equivalent to bitstring[1].
char	An 8-bit field used as a character. Same as string[1].

Table 6-5. Rez type specifications (continued)

Specification	*Description*
string[*n*]	A plain string (with no preceding character count and no termination character). The value of *n* is the length of the string, in bytes. If you precede the word "string" with the word "hex" (for example, hex string[32]), Rez displays the string as a string of hexadecimal characters *n* characters long.
pstring[*n*]	A Pascal-style string (a string preceded by a byte specifying the length of the string). The length of the string, in bytes, is the value of *n* plus 1. A Pascal string can be no more than 255 characters long; if it is too long, Rez displays an error and truncates the string.
wstring[*n*]	A string that can be up to 65,535 characters long. The length of the string, in bytes, is the value of *n* plus 2.
cstring[*n*]	A C-style string (a string followed by a trailing null used as a termination character). The length of the string is the value of *n* minus one.
point	A pair of two 16-bit integers. The first integer describes the point's vertical (y) coordinate. The second integer describes the point's horizontal (x) parameter.
rect	A series of four 16-bit integers. Each pair of integers is interpreted as a point (see "point"). The first point describes the coordinates of the upper left-hand corner of a rectangle. The second point describes the coordinates of the lower right-hand corner of the rectangle.
fill *fillSize* [*n*]	Fills a specified number of bits in the data stream with zeros. The *fillSize* parameter must be one of the following constants: bit (one bit) nibble (four bits) byte (eight bits) word (16 bits) long (32 bits) The length of the zero fill generated by the fill specification is the value of *n* times the number of bits specified by the *fillSize* parameter.

Table 6-5. Rez type specifications (continued)

Specification	*Description*
align *fillSize* [*n*]	Fills a specified number of bits in the data stream with zeros, and then pads the end of the filled area with more zeros until a specified boundary is reached. The align specification can terminate on a 4-bit, 8-bit, 16-bit, or 32-bit boundary. The *fillSize* parameter must be one of the following constants: nibble (four bits) byte (eight bits) word (16 bits) long (32 bits) The length of the zero fill generated by the align specification is the value of *n* times the number of bits specified by the *fillSize* parameter. The align specification affects all data from the point where it is specified until the next align statement.
switch	Used in resource descriptions that contain case statements; used much like the switch statement in C, or the CASE statement in Pascal.
array	Declares a list of fields (delimited by curly brackets) that are repeated for each element of an array. The array specification is explained in more detail under the next heading.

Arrays

The Rez language includes a provision for placing arrays in resource descriptions. This is the definition of an array:

```
array [ arrayName | '['length']'] {arrayList};
```

The *arrayList* parameter is a list of type specifications. It can be repeated zero or more times. Each element that makes up the array list is separated from the next by a comma and a space. Either the *arrayName* parameter or the *length* parameter may be specified in a resource description. The *arrayName* parameter identifies the array list.

The declaration of an array may be preceded by the keyword wide, as follows:

```
wide array [ arrayName | '['length']'] {arrayList};
```

If the keyword wide is used, DeRez uses a more compact display format, but this does not affect Rez. In complex resources, the use of the keyword wide is preferred because it can considerably reduce the amount of data required by DeRez.

An array contains a list of fields, delimited by curly brackets, that are repeated for each element of the array. The general form of an array description is:

```
array {
    /* definitions of fields */
};
```

Once an array is described, it can be nested in any other resource description, for example,

```
type 'RSRC' {
    literal longint;     /* resource type */
    wide array {
        integer;         /* each resource ID */
    };
};
```

This type of array is fine for simple data structures, but sometimes it is useful to define the number of elements in an array so that the program using the resource can determine how many elements they are, and so that DeRez can compile the resource more easily.

To keep track of how many variables there are in an array, you can use the Rez function $$CountOf, which returns the number of elements in an array. In the following example, the $$CountOf function returns the number of strings in the array StrArray:

```
type 'STR#' {/* a string list resource */
    integer = $$CountOf (StrArray);
    wide array StrArray {
        pstring;
    };
};
```

Structured Data Types

When you write a resource definition, you can use constants and symbolic names to assign values to fields in the resource you have defined. For example, suppose you have defined a resource type 'HORS', as follows:

```
type 'HORS'{/* declaring a resource of type 'HORS' */
    integer;
    string;
    byte off = 0, on = 1;
    integer = 36;
};
```

According to this definition, a 'HORS' resource has four fields:

- An integer that can hold any integer value.
- A C-style string.
- A byte that can be set to either 0 or 1 by using the symbolic name "on" or "off."
- An integer that has a value of 36. Since this value is defined in the 'HORS' resource template, it doesn't have to appear in subsequent resource descriptions that use the 'HORS' resource template.

Once you have defined a resource of type 'HORS', you can place a 'HORS' resource in a resource description file by simply filling in the fields provided in the resource definition. You could do that by typing a resource statement such as this:

```
resource 'HORS' (280) {
    56;
    "This is the racehorse resource.";
    off;
};
```

The first line of this example says that a resource of type 'HORS' is being described, and that its resource ID is 280. The three lines that follow, enclosed in curly brackets, fill in the first three fields of the 'HORS' resource template. The first field is an integer, 56; the second is a string; and the third is a bit that can be set or cleared using the symbolic name "on" or "off." The last field in the 'HORS' resource type does not appear in the preceding resource description because it is predefined in the resource type definition; it is always 36.

Once you have defined a resource type, you can create as many resources of that type as you like, placing any legal value that you like in each field. For example, you could describe another resource of type 'HORS' this way:

```
resource 'HORS' (281) {
    232;
    "This is a horse of a different color.";
    on;
};
```

▶ The resource Statement

Now that we have examined the trivial resource type 'HORS', let's examine a more serious example. Let's assume that you have defined the resource type 'RECT' as a rectangle. Once you have done that, you can describe a rectangle resource using a statement such as:

```
resource 'RECT' (rTitleBox, preload, purgeable) {
    {5,5,24,250}
};
```

Note that in the resource description, the coordinates of a rectangle are used in place of the type specifier "rect" in the resource definition.

That's still a very simple example of a resource description. Listing 6-1 is a more complex example—the resource type 'WIND', the official window resource type defined in the Types.r interface file.

Listing 6-1. Definition of the 'WIND' resource

```
type 'WIND' {
    rect;           /* window's bounds rectangle */
    integer     documentProc, dBoxProc, plainDBox,
                /* predefined types */
                altDBoxProc, noGrowDocProc,
                zoomProc = 8, rDocProc = 16;
    byte        invisible, visible; /* window visible? */
                fill byte;
    byte        noGoAway, goAway; /* close box? */
                fill byte;
    unsigned hex longint; /* refCon (see Inside
                Macintosh) */
    pstring     Untitled = "Untitled"   /* window
                title */
};
```

In the next-to-last line of Listing 6-1, note the use of the words "unsigned," "hex," and "longint." These words come from a list of predefined constants that can be used as values in resource type definitions. The complete list is shown in Table 6-6.

Table 6-6. Constants used in resource type definitions

Constant	*Meaning*
bitstring[length]	Bitstring of length bits (maximum 32)
byte	Byte (8-bit) field
integer	Integer (16-bit) field
longint	Long integer (32-bit) field
hex	Hexadecimal value
decimal	Decimal value
octal	Octal value
binary	Binary value
literal	Literal data

To place a 'WIND' resource in a resource description file, you could write a window description like the one in Listing 6-2.

Listing 6-2. A window resource

```
resource 'WIND' (200, "My Window," appheap, preload)
    { /* heading */
    {20,20,120,300},     /* window's bounds rectangle */
    documentProc,        /* document window */
    visible,             /* visible window */
    goAway,              /* has close box */
    0,                   /* refCon (see Inside Macintosh) */
    "Sample Window"      /* window title */
};
```

Compare the resource type definition in Listing 6-1 with the resource description in Listing 6-2, and you'll see that the description merely fills in the fields defined in the definition, substituting commas for semicolons at the ends of fields, and using symbolic names for field values where appropriate.

▶ The data Statement

When you use a data statement in a resource description file, Rez interprets the information that defines the resource as raw data. You can use the data in any way you like in an application.

For example, this statement describes a block of data that has been defined in an application or an interface file as data type 'DORF':

```
data 'DORF' (128, preload) {
     $"30F5 90F4 E59C DEF4 68A0"
};
```

You can place any kind of data you like in a data resource; for example, you could describe a data resource as a bit map of a screen. Most data resources are much longer than the one shown in the preceding example.

▶ include Statement

The Rez statement include—which should be distinguished from the preprocessor directive #include—can be used to merge previously compiled resources from the resource fork of another file into the resource definition file currently being written. When you use include to combine another resource fork with the one you are creating, you can use all of the resources in the other file, or you can bring in only resources of a specified type. If you wish, you can also redefine the types of the resources you are importing. The statement

```
include "fileB";
```

imports all the resources in the resource fork of fileB into the resource description file that you are writing. But the statement

```
include "fileB" 'ICON';
```

imports only the 'ICON' resources in fileB's resource fork.

With the keyword include, you can use the words "not" and "as" to specify what types of resources you want to import and how you want their types redefined. For instance, the statement

```
include "fileB" not 'ICON';
```

brings into your file only those resources in fileB that are *not* icons. The statement

```
include "fileB" 'typeA' as 'typeB';
```

imports resources in file B that are defined as type A, but defines their type as type B.

If you wanted to import only one resource from another file, you could write a statement like this:

```
include "fileB" 'ICON' (128);
```

which would import only the icon with an ID number of 128 from fileB.

MPW has a set of variables that you can use with the word "as" to modify the information in include statements. These variables are:

Variable	*Meaning*
$$Type	Type of resource from include file.
$$ID	ID of resource from include file.
$$Name	Name of resource from include file.
$$Attributes	Attributes of resource from include file.

When a variable from the preceding list appears in an include statement following the word "as," the information specified in the resource fork being imported remains the same in the resource description file being written. Statements that use these variables are written in this format:

```
include "fileName" rsrcType (rsrcName | (ID[:ID)])
    as rsrcType (ID [, rsrcName] [, attributes...]);
```

In the first line of this description, the colon between the two *ID* parameters specifies a range of resource ID numbers. Each resource of the specified type with an ID that falls into the specified range is imported. For example, the statement

```
include "fileB" 'DRVR' (0:32)
    as 'DRVR' ($$ID, $$Name, $$Attributes | preload);
```

imports all driver resources (type 'DRVR') with resource IDs ranging from 0 through 32. The ID numbers and names of the imported resources remain unchanged, but the bitwise OR operator | adds the attribute "preload" to the attributes field of each imported resource. (Resource attributes are described later in this chapter.)

The read Statement

The read statement merely reads the data fork from a file and writes it as a resource to the resource description file being created. For example, the statement

```
read 'STR ' (218, "My String," sysheap, preload) ∂
    "OtherFile";
```

reads a string resource from the resource fork of the OtherFile file and incorporates it into the resource description file being written. In this case, the string resource being imported has an ID number of 218, is named "My String," and has an attribute field with the sysheap and preload attributes set. (There's more about resource attributes later in this chapter.)

The change Statement

You can use the change statement to change a resource's vital information. A resource's vital information includes its resource type, resource ID, name, attributes, or any combination of these.

The change statement is often used with the | operator (bitwise OR) to make sure that an attribute is set. For instance, this MPW shell command sets the protected bit to 1 on all code resources in the TestDA file:

```
echo "change 'CODE' to $$Type ($$ID,$$Attributes | 8);" ∂
    | rez -a -o TestDA
```

The delete Statement

You can delete a resource without launching ResEdit by using the delete statement. This is an example of a shell command that deletes a 'HORS' resource from a file called DobbinFile:

```
delete "delete 'HORS';" | rez -a -o "DobbinFile"
```

Labels

In complex resource descriptions, such as the descriptions of QuickDraw resources, labels are sometimes used to permit the accessing of data at specified locations within a resource. When a label is used in a resource description, it is followed by a colon. For example, in the resource definition

```
type 'lucy' (192) {
    cstring
stringEnd:
    integer = endOfString;
};
```

the label *stringEnd* could be used to locate the end of the C string that precedes it.

Variables and Functions

The Rez compiler has a set of variables and functions that contain or return commonly used values. All Rez variables and functions are written as strings preceded by two dollar signs, as follows:

```
$$Date
```

In the resource description language, variables and functions can be used in structured constructs and arrays to stand for a field in a resource currently being processed. They are used only in complex resource descriptions.

In super-turbocharged applications, the include statement, described earlier in this chapter, often makes use of variables and functions. Listing 6-13, later in this chapter, shows how variables and functions can be used in a resource description.

Table 6-7 is a list of variables and functions that have string values. Table 6-8 lists variables and functions that have numeric values.

Table 6-7. Variables and functions with string values

Variable or Function	*Value*
$$Date	Returns the current date.
$$Format ("*formatString*", *arguments*)	Works like the #printf directive, but returns a string rather than printing to standard output.
$$Name	The name of the resource currently being created, included, deleted, or changed (depending on whether a resource, include, delete, or change statement is being processed).
$$Resource ("*fileName*", '*type*', *ID* \| "*resourceName*")	Given the ID number or name of a resource, $$Resource returns the resource type as a string.
$$Shell ("*stringExpr*")	Given a shell variable {*stringExpr*}, $$Shell returns the variable's current value. The curly brackets that delimit the variable are omitted; double quotation marks are substituted.
$$Time	Returns the current time.
$$Version	Returns the version number of the Rez compiler being used.

Table 6-8. Variables and functions with numeric values

Variable or Function	*Value*
$$Attributes	Contains attributes of resource currently being created, included, deleted, or changed (depending on whether a resource, include, delete, or change statement is being processed).
$$BitField(*label*, *starting Position numberOfBits)*	Given a starting position and a label in a resource description, $$BitField returns the number of bits in the bitstring found.

Table 6-8. Variables and functions with numeric values (continued)

Variable or Function	*Value*
$$Byte	Given a label in a resource description, $$Byte returns the byte found at the label specified.
$$Day	Returns the current day as a number ranging from 1 through 31.
$$Hour	Returns the current hour as a number ranging from 0 through 23.
$$ID	Returns the ID of the resource currently being created, included, deleted, or changed (depending on whether a resource, include, delete, or change statement is being processed).
$$Long	Given a label, $$Long returns the longword found at the label specified.
$$Minute	Returns the current minute as a number ranging from 0 through 59.
$$Month	Returns the current month as a number ranging from 1 through 12.
$$PackedSize (*start*, *rowBytes*, *rowCount*)	Given an offset into the resource currently being processed (*start*) and two integers (*rowBytes* and *rowCount*), $$PackedSize calls the Toolbox routine UnpackBits RowCount times and returns the unpacked data found at *start*.
$$ResourceSize	Returns the size, in bytes, of the resource currently being created, included, deleted, or changed (depending on whether a resource, include, delete, or change statement is being processed).
$$Second	Returns the current second as a number ranging from 0 through 59.

Table 6-8. Variables and functions with numeric values (continued)

Variable or Function	*Value*
$$Type	Returns the size, in bytes, of the resource currently being created, included, deleted, or changed (depending on whether a resource, include, delete, or change statement is being processed).
$$Weekday	Returns the current day of the week as a number ranging from 1 (Sunday) through 7 (Saturday).
$$Word(*label*)	Given a label, returns the word at the label specified.
$$Year	Returns the current year.

▶ Arithmetic and Logical Expressions

You can perform many kinds of arithmetical and logical operations in the Rez language. All arithmetic is performed as 32-bit signed arithmetic. The basic constants used in the Rez language are listed in Table 6-9. The language's arithmetic and logical operators were listed in Table 6-2.

Table 6-9. Numeric constants in the Rez language

Numeric Type	*Form*	*Meaning*
Decimal	nnn...	Signed decimal constant between 4,294,976,295 and 2,147,483,648.
Hexadecimal	0xhhh...	Signed hexadecimal constant between 0x7FFFFFFF and 0x80000000.
Octal	0ooo...	Signed octal constant between 017777777777 and 020000000000.
Binary	0bbbb...	Signed constant between 0b10000000000000000000000000000000 and 0b11111111111111111111111111111111.
Literal	'aaaa'	May contain one to four characters (printable ASCII characters or escape characters.

The Rez compiler treats letters and numbers in the same way; both letters and numbers are interpreted as numeric values. However, a letter within single quotation marks is recognized as a literal, as shown in Table 6-9, and a value within double quotation marks is recognized as a character. For example, since the letter "A" has an ASCII value of 65, 'A' (in single quotes) is interpreted by the compiler as the number 65, but "A" (in double quotes) is interpreted as the character "A". However, both 'A' and "A" are represented in memory by the bitstring 01000001, the binary equivalent of 65. Thus, 'A' = 65, 'B' = 66, and 'A'+1 = 66.

For an illustration of how arithmetic and logical expressions are used in resource descriptions, see Listing 6-13.

▶ The Rez Command

When you have written a resource file, you can compile the file—that is, convert it from source code into object code—using the MPW command Rez. You can then use the MPW Linker to link your resource file with the object code for the rest of your program. The syntax of the Rez command is:

```
Rez [sourceFile...] [option...] [destFile...]
```

where *sourceFile* is the name of a resource description file to be read by Rez, and *destFile* is the name of the object file associated with the resource fork. For example, the command

```
Rez Types.r Creation.r -o Creation
```

generates a resource fork for a file named Creation, based on information provided in the interface file Types.r and the resource definition file Sample.r.

Options that can be used with the Rez command are listed here.

Option	*Meaning*
-a[ppend]	Merge resource into output resource file.
-align word | longword	Align resource to word or longword boundaries.
-c[reator] creator	Set output file creator.
-d[efine] name[=value]	Equivalent to #define macro [value].
-i[nclude] pathname	Path to search when looking for #include files.
-m[odification]	Don't change the output file's modification date.

Option	*Meaning*
-o file	Write output to file (default is a file named rez.out).
-ov	OK to overwrite protected resources when appending.
-p	Write progress information to diagnostics.
-rd	Suppress warnings for redeclared types.
-ro	Set the mapReadOnly flag in output.
-s[earch] pathname	Path to search when looking for #include resources.
-t[ype] type	Set output file type.
-u[ndef] name	Equivalent to #undef name.

Status codes returned by the Rez command are as follows.

Error	*Meaning*
0	No errors.
1	Error in parameters.
2	Syntax error in file.
3	I/O or program error.

How the Rez Command Works

Rez compiles the resource fork of a file in accordance with a resource description file written in a special language and stored as a text file with the extension .r. Resource description files recognized by Rez have the same format as resource files created with DeRez, MPW's resource decompiler, which is described in the next section.

The information that Rez uses to build a resource file can come not only from the data in a resource description file, but also from other text files that are associated with the resource description file via #include and read directives. The #include directive can be used to associate a resource description file with definitions of constants and also with other resource description files.

Standard resource type declarations that can be used with the Rez command are provided in several interface files in the MPW directory {RIncludes}. Interface files that can be used with Rez include Types.r, SysTypes.r, MPWTypes.r, and Pict.r. Features of the Rez command include macro processing, full expression evaluation, and built-in functions and system variables.

Although the Rez command can be used by itself, it is more commonly used in makefiles: MPW scripts designed to compile and link programs automatically. Procedures for writing and using makefiles are described in Chapter 8.

▶ The DeRez Command

The DeRez command is a mirror image of the Rez command. The input to the DeRez command is the resource fork of an object file. DeRez decompiles the resource fork and creates a text file, or resource description file, that can be read by Rez. Unless redirection is used, DeRez writes a resource description file to standard output, normally the screen. The syntax of the DeRez command is:

```
DeRez [option...] rsrcFile [rsrcDescriptionFile...]
```

For example, the command

```
DeRez Creation Creation.r
```

writes a resource description file to standard output, using the file Creation.r for resource descriptions.

Options that can be used with the DeRez command are listed here.

Option	*Meaning*
-c[ompatible]	Generate output compatible with Rez Version 1.0.
-e[scape]	Don't escape chars < $20 or > $D8.
-d[efine] name[=value]	Equivalent to #define name [value].
-i[nclude] pathname	Search this path when looking for #include files.
-m[axstringsize] n	Write strings *n* characters per line.
-only typeExpr*	Process only resources of this type.
-p[rogress]	Write progress information to diagnostics.
-rd	Suppress warnings for redeclared types.
-s[kip] typeExpr	Skip resources of this type.
-u[ndef] name	Equivalent to #undef name.

*A typeExpr may have one of these forms:

type	"'type'(id:id)"
'type'(id)"	"'type'(..."name...")"

Status codes returned by the DeRez command are as follows.

Error	*Meaning*
0	No errors.
1	Error in parameters.
2	Syntax error in file.
3	I/O or program error.

How the DeRez Command Works

As input, the DeRez command takes the name of an object file that has a resource fork. From the resource fork, it creates a resource description file. Unless redirection is used, the resource description is written to standard output.

A resource description file is made up of type declarations written in the format recognized by MPW's resource compiler, Rez. The type declarations are defined in the MPW interface files Types.r and SysTypes.r.

If you used the output of a DeRez command as input to Rez, with the same resource description files, DeRez would theoretically produce a resource fork identical to the one that was originally input to DeRez. However, DeRez can become confused by the contents of complex resource forks, particularly if they contained user-defined resources. If DeRez cannot create a resource description that matches a given resource, it writes a string of data that matches the resource that it was unable to describe.

▶ The ResEqual Command

ResEqual is an MPW command that compares the resources in two files and writes their differences to standard output. Its syntax is

```
ResEqual [-p] File1 File2
```

In a ResEqual command, the File1 and File2 parameters are the files being compared. If you use the -p option, ResEqual writes progress information to diagnostic output. For example, the command

```
ResEqual Creation Creation.rsrc
```

compares the resources in the Creation and Creation.rsrc files and writes the results to standard output.

Status codes returned by the ResEqual command are as follows.

Error	*Meaning*
0	Resources match.
1	Parameter or option error.
2	Files don't match.

ResEqual compares two files and confirms that:

- Each file contains resources of the same type and identifier as the other.
- The size of the resources with the same type and identifier are the same in both files.
- The contents of the resource forks of the compared programs are the same.

If a mismatch is found, ResEqual reports it and then continues the comparison until it finishes comparing the files. If more than ten differences are detected in the same resource, the rest of the resource is skipped and ResEqual moves on to the next resource.

By the Way ▶

A Rez by Any Other Name. You may have noticed that the word "resource" is sometimes abbreviated "Rez," and sometimes abbreviated "Res," in MPW commands. Here's why: The ResEqual command was written in Pascal and was named after ResEdit. But Rez, DeRez, and RezDet were written in C, and were named separately. Apple insiders say that whole meetings were spent arguing about the spellings of these tools. But, in the end, as Daniel K. Allen reports in his book *On Macintosh Programming: Advanced Techniques* (Addison-Wesley, 1989), "the struggle for uniformity ended in a stalemate."

▶ The RezDet Command

You can check to see if a resource file contains any errors by executing MPW's RezDet command. Its syntax is:

```
RezDet [option...] file...
```

For example, the command

```
RezDet -q Creation || Delete Creation
```

deletes the Creation file if the resource fork is damaged.

Options that can be used with RezDet are listed here.

Option	*Description*
-b[ig]	Read resources one at a time, not all at once.
-d[ump]	Write information plus headers, lists, etc.
-l[ist]	Write list of resources with minimum information.
-q[uiet]	Don't write any output, just set {Status}.
-r[awdump]	Write -dump information plus contents.
-s[how]	Write information about each resource.

If you do not specify any option, RezDet investigates the resource fork of each file for damage or inconsistencies. The specified files are read and checked one by one. Output is generated according to the options specified.

You should use no more than one of the following options: -quiet, -list, -show, -dump, and -rawdump.

Status codes returned by the RezDet command are as follows.

Error	*Meaning*
0	No errors detected.
1	Invalid options or no files specified.
2	Resource format error detected.
3	Fatal error—an I/O or program error was detected.

Specifically, RezDet checks for these conditions:

- Is the resource fork at least the minimum size? (There must be enough bytes to read a resource header.)
- Is each record in the resource data list used once and only once? The last data item in a resource should end exactly where the data list ends.
- Are there any duplicate types in the resource type list?
- Does each item in the resource type list contain at least one reference?

- Does each sequence of referenced items start where the previous resource type item's reference list ended? Is each item in the reference list pointed to by one and only one resource type list item?
- Does each name in the name list have one and only one reference? Does the last name point outside the name list? (It shouldn't.)
- Are there are any duplicate names in the name list? (Duplicate names generate an advisory warning rather than an error, and they don't even signal a warning if the -s, -d, or -r option is selected.)
- Do all names have a nonzero length?
- Are Bits 7 (Unused), 1 (Changed), or 0 (Unused) set in the resource attributes field? (They shouldn't be.)
- Does each item on the reference list point to a valid data item? Does each item either have a name list offset of -1 or point to a valid name list offset?
- Is there any space (or overlap) between the header, the resource data list, and the resource map? There should not be any bytes between the EOF and the end of the resource map.

▶ The Structure of a Resource

Many kinds of resources—such as strings, icons, and fonts—have formats that are defined as templates in the MPW interface file Types.r. For example, an icon is defined as a 32-by-32 bit image, a string is defined as a Pascal string, and a bitmapped font is stored as a large bit image containing a set of characters.

The simplest resource type is 'STR ' (note the space before the second quotation mark), which is defined in Types.r as a Pascal-format string. In the Types.r interface file, an 'STR ' resource is defined as

```
type 'STR ' {
        pstring;    /* String */
};
```

This is what the source code for a 'STR ' resource might look like in a typical resource file:

```
resource ('STR ', 128) {
    "The racehorses are managed by the racehorse manager."
};
```

A longer string could be written like this:

```
resource ('STR ', 129) {
   "It was a dark and stormy night: The rain fell in "
   "torrents except at occasional intervals, when it was "
   "checked by a violent gust of wind which swept up the "
   "streets (for it was in London that our scene lies), "
   "rattling along the housetops, and fiercely agitating "
   "the scanty flame of the lamps that struggled against "
   "the darkness."
};
```

In the headings of the preceding examples, the numbers 128 and 129 are resource IDs, not character counts; Rez calculates the length of a Pascal string automatically. Resource IDs are described later in this chapter.

▶ Fields in Resource Templates

As you saw earlier in this chapter, some resources have templates that are divided into fields, much like the data structures that are used in Pascal and C. In a menu resource, there are fields for the menu's title, the text of each item listed under the menu, information that specifies whether the menu and its items should be enabled or disabled, and any characters or special characters that may appear alongside each menu item.

A Macintosh program typically includes a menu-bar resource, plus a resource for each menu on the menu bar. A menu-bar resource lists the names of the menus that appear on the menu bar, and each menu resource contains a list of menu items.

In the Types.r interface file, an 'MBAR' resource is defined as shown in Listing 6-3, and a 'MENU' resource is defined as shown in Listing 6-4.

The meanings of the fields in Listings 6-3 and 6-4 are described in more detail in Chapter 3, and in the Menu Manager chapter of *Inside Macintosh*.

Listing 6-3. Definition of an 'MBAR' resource

```
type 'MBAR' {
   integer = $$CountOf(MenuArray);  /* Number of menus */
   wide array MenuArray{
           integer;                 /* Menu resource ID */
   };
};
```

Listing 6-4. Definition of a 'MENU' resource

```
type 'MENU' {
    integer;                                /* Menu ID */
    fill word[2];                           /* Menu size placeholders */
    integer textMenuProc = 0;               /* Menu DefProc ID */
    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;
};
```

Listing 6-5 shows what the source code for a menu bar might look like in a typical resource file. The source code for one of the menus on the menu bar is shown in Listing 6-6.

Listing 6-5. A menu bar resource

```
resource 'MBAR' (rMenuBar, preload) {
    { mApple, mFile, mEdit, mWindows }; /* 4 menus */
};
```

Listing 6-6. A menu resource

```
resource 'MENU' (mFile, preload) {
    mFile, textMenuProc,
    0b1111111111111111111111111111100,
    enabled, "File",
    {
        "New",
            noicon, "N", nomark, plain;
        "Open",
            noicon, "O", nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Close",
            noicon, "W", nomark, plain;
        "Save",
            noicon, "S", nomark, plain;
        "Save As…",
            noicon, nokey, nomark, plain;
        "Revert",
            noicon, nokey, nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Page Setup…",
            noicon, nokey, nomark, plain;
        "Print…",
            noicon, nokey, nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Quit",
            noicon, "Q", nomark, plain
    }
};
```

The third line in Listing 6-6—the long binary number—shows which menu items are enabled and which are disabled in the menu being described. To decrypt the number, you read it from right to left—that is, from the last binary digit to the first. In Listing 6-6, since the last two digits in the number are zeros, the first two menu items—New and Open—are disabled when the program begins. Since all the other digits in the number are ones, all the other menu items are enabled when the program starts. Note that when you examine the items under a menu to set this value, each dash that is used to display a horizontal line in the menu counts as a menu item.

The other fields in a menu resource description are easy to figure out. The constants noicon, nokey, nomark, and plain mean that the item being described has no icon, no check mark, and no keyboard equivalent, and is displayed in a plain type style. Keyboard equivalents are enclosed in quotation marks, and a hyphen inside quotation marks displays a horizontal line that separates menu items.

▶ The SIZE Resource

One of the most important resources in a Macintosh program is one called the 'SIZE' resource. There is a 'SIZE' resource in the resource fork of every program that is designed to work with MultiFinder. The 'SIZE' resource tells MultiFinder the size of the memory partition to use when running your application. It also provides a host of important facts about it—whether it accepts suspend and resume events, whether it runs in the background, whether it is MultiFinder-aware, and so on.

Listing 6-7 shows how the 'SIZE' resource is defined in the Types.r interface file. A typical 'SIZE' resource is shown in Listing 6-8.

Listing 6-7. Definition of the 'SIZE' resource

```
type 'SIZE' {
        boolean  dontSaveScreen, /* for SWITCHER          */
                 saveScreen;     /*    compatibility      */
        boolean  ignoreSuspendResumeEvents, /* suspend-resume     */
                 acceptSuspendResumeEvents;
        boolean  enableOptionSwitch, /* for SWITCHER      */
                 disableOptionSwitch;     /*    compatibility     */
        boolean  cannotBackground,
                 canBackground; /* Can properly use back-
                          ground null events    */
        boolean  notMultiFinderAware, /* activate/deactivate      */
                 multiFinderAware;    /* on resume/suspend   */
        boolean  backgroundAndForeground, /* Application does not */
                 onlyBackground;      /* have a user interface*/
        boolean  dontGetFrontClicks, /* Get mouse down/up    */
                 getFrontClicks; /* when suspended          */
        boolean  ignoreChildDiedEvents,    /* Apps use this. */
                 acceptChildDiedEvents;    /* Debuggers use this. */
        boolean  not32BitCompatible,       /* Works with 24bit addr*/
```

```
                is32BitCompatible;       /* Works with 24 or 32 */
                            /* bit addresses   */
#undef reserved
        boolean  reserved;   /* These seven bits are  */
        boolean  reserved;   /* reserved.  Set them   */
        boolean  reserved;   /* to "reserved". When   */
        boolean  reserved;   /* we decide to define   */
        boolean  reserved;   /* a new flag, your      */
        boolean  reserved;   /* old resource will     */
        boolean  reserved;   /* still compile.        */

        /* Memory sizes are in bytes */
        unsigned longint;    /* preferred mem size    */
        unsigned longint;    /* minimum mem size      */

// If we ever define one of the seven reserved bits above, the
// "reserved" enumeration wouldn't appear on the newly defined bit.
// By defining "reserved" below, old SIZE declarations
// still compile.

#define reserved     false
};
```

Listing 6-8. Description of a 'SIZE' resource

```
resource 'SIZE' (-1) {     /* We have here a MultiFinder-
                            aware application */
    dontSaveScreen,
    acceptSuspendResumeEvents,
    enableOptionSwitch,
    canBackground,
    multiFinderAware,
    backgroundAndForeground,
    dontGetFrontClicks,
    ignoreChildDiedEvents,
    not32BitCompatible,
    reserved, reserved, reserved, reserved, reserved,
          reserved, reserved,
    96*1024,
    64*1024
};
```

▶ Resource Specifications

Although different kinds of resources have different formats, all resources have certain features in common. For example, every resource begins with a heading called a resource specification. The information that follows the resource specification—the data that defines the content of the resource—is called resource data.

A resource specification always contains a four-character string called a resource type and a 16-bit number called a resource ID. A resource specification can also include a resource name, written as a string, and set of resource attributes, written as integers.

In the menu resource shown in Listing 6-6, the resource type is 'MENU', and the resource name is the *mFile* variable, which is defined elsewhere in the program. The menu resource shown in Listing 6-6 has a resource attribute defined as "preload." Resource attributes are explained in more detail under the following headings.

Resource Types

The first element in a resource specification is a resource type: a series of four ASCII characters from 0 to 255, enclosed in single quotation marks. The most commonly used resource types are written in all uppercase or all lowercase letters.

The format of a resource is specified by its resource type. For example, a menu resource has the resource type 'MENU', a string resource has the resource type 'STR ', and a dialog resource has the resource type 'DLOG.'

In MPW Pascal, a resource type is defined as:

```
TYPE ResType = PACKED ARRAY[1..4] OF CHAR;
```

Table 6-10 is a list of predefined resources identified by resource type.

Table 6-10. Predefined Macintosh resources

Type	*Description*
'ALRT'	Alert template
'ADBS'	Apple Desktop Bus service routine
'BNDL'	Bundle
'CACH'	RAM cache code
'CDEF'	Control definition function
'CNTL'	Control template
'CODE'	Application code segment
'CURS'	Cursor
'DITL'	Item list in a dialog or alert
'DLOG'	Dialog template
'DRVR'	Desk accessory or other device driver
'DSAT'	System startup alert table
'FKEY'	Command-Shift-number routine
'FMTR'	3 1/2-inch disk formatting code
'FOND'	Font family record
'FONT'	Font
'FREF'	File reference
'FRSV'	IDs of fonts reserved for system use
'FWID'	Font widths
'ICN#'	Icon list
'ICON'	Icon
'INIT'	Initialization resource
'INTL'	International resource
'INT#'	List of integers owned by Find File
'KCAP'	Physical layout of keyboard (used by Key Caps desk accessory)
'KCHR'	ASCII mapping (software)
'KMAP'	Keyboard mapping (hardware)
'KSWP'	Keyboard script table
'LDEF'	List definition procedure
'MBAR'	Menu bar
'MBDF'	Default menu definition procedure
'MDEF'	Menu definition procedure

Table 6-10. Predefined Macintosh resources (continued)

Type	*Description*
'MENU'	Menu
'MMAP'	Mouse tracking code
'NBPC'	AppleTalk bundle
'NFNT'	128K ROM font
'PACK'	Package
'PAT '	Pattern (The space is required.)
'PAT#'	Pattern list
'PDEF'	Printing code
'PICT'	Picture
'PREC'	Print record
'PRER'	Device type for Chooser
'PRES'	Device type for Chooser
'PTCH'	ROM patch code
'RDEV'	Device type for Chooser
'ROvr'	Code for overriding ROM resources
'ROv#'	List of ROM resources to override
'SERD'	RAM Serial Driver
'SICN'	Script symbol
'STR '	String (The space is required.)
'STR#'	String list
'WDEF'	Window definition function
'WIND'	Window template
'atpl'	Internal AppleTalk resource
'bmap'	Bitmaps used by the Control Panel
'boot'	Copy of boot blocks
'cctb'	Control color table
'cicn'	Color Macintosh icon
'clst'	Cached icon lists used by Chooser and Control Panel
'clut'	Color look-up table
'crsr'	Color cursor
'ctab'	Used by the Control Panel
'dctb'	Dialog color table
'fctb'	Font color table

Table 6-10. Predefined Macintosh resources (continued)

Type	*Description*
'finf'	Font information
'gama'	Color correction table
'ictb'	Color table dialog item
'insc'	Installer script
'itl0'	Date and time formats
'itl1'	Names of days and months
'itl2'	International Utilities Package sort hooks
'itlb'	International Utilities Package script bundles
'itlc'	International configuration for Script Manager
'lmem'	Low memory globals
'mcky'	Mouse tracking
'mctb'	Menu color information table
'mitq'	Internal memory requirements for MakeITable
'mppc'	AppleTalk configuration code
'nrct'	Rectangle positions
'pltt'	Color palette
'ppat'	Pixel pattern
'snd '	Sound (The space is required.)
'snth'	Synthesizer
'wctb'	Window color table

Defining Your Own Resource Types

If you want to use a resource type that is not predefined by the Macintosh system, you can design a resource of your own. For instance, there is no predefined resource structure for a rectangle. But you could declare a resource type called 'RECT', define its structure as the structure of a rectangle, and then use rectangles as resources in the program. The definition of a rectangle resource is shown in Listing 6-9, and source code that creates a rectangle resource of type 'RECT' is shown in Listing 6-10.

Listing 6-9. Defining a resource type

```
type 'RECT' {
        rect;
};
```

Listing 6-10. Source code for a user-defined resource

```
resource 'RECT' (rTitleBox, preload, purgeable) {
    {5,5,24,250}
};
```

When you design your own resources, you can give them type names written in either uppercase or lowercase letters—in fact, you can use any ASCII characters from 0 to 255. But, remember that when it interprets resource types, Rez *is* case sensitive. Thus the resources 'wxyz', 'WXyz', and 'WXYZ' are all different!

Resource IDs

The second element in a resource specification is a 16-bit number called a resource ID. In a resource file, every resource of the same type must have a unique resource ID. However, a resource of a given type may have the same resource ID as a resource of a different type. For example, a resource fork could include a menu resource with an ID of 128 and a dialog resource with an ID of 128. But a resource file should not contain two dialog resources with an ID of 128. If it does, the result of trying to access either of them is unpredictable.

In some cases, resources are associated with each other by their resource IDs. For example, the next-to-last field in a dialog resource is always the resource ID of a second resource: a list of dialog items owned by the dialog. Alert dialogs, which have the resource type 'ALERT', are also associated with lists of dialog items.

A list of dialog items stored as a resource has a resource type of 'DITL'. When the next-to-last field of a dialog resource contains the resource ID of a specified 'DITL' resource, the items in the 'DITL' resource are placed inside the dialog with which they are associated each time the dialog is drawn.

The alert resource shown in Listing 6-11 has an item list with a resource ID identified as the value of the *rAboutAlert* variable (the variable is defined elsewhere in the application). The alert is associated with an item list with the same ID number, as you can see by looking at the second line of Listing 6-11.

The list of items associated by the dialog is shown in Listing 6-12. The resource ID assigned to the list of items is, of course, rAboutAlert—the same ID assigned to the alert with which it is associated.

Listing 6-11. An alert resource

```
resource 'ALRT' (rAboutAlert, purgeable) {
    {40+30, 20+30, 160+30+20, 296+30}, rAboutAlert, {
        OK, visible, silent;
        OK, visible, silent;
        OK, visible, silent;
        OK, visible, silent
    };
};
```

Listing 6-12. A 'DITL' resource

```
resource 'DITL' (rAboutAlert, purgeable) {
    { /* array DITLarray: 5 elements */
        /* [1] */
        {88+20, 184, 108+20, 264},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {8, 8, 24, 274},
        StaticText {
            disabled,
            "                          Creation"
        },
        /* [3] */
        {32, 8, 48, 237+20},
        StaticText {
            disabled,
            "    An All-Purpose Macintosh Program"
        },
        /* [4] */
        {56, 8, 72, 237+20},
        StaticText {
            disabled,
            "                    By Mark Andrews"
```

Listing 6-12. A 'DITL' resource (continued)

```
        },
        /* [5] */
        {80+20, 24, 112+20, 167},
        StaticText {
            disabled,
            "Copyright © 1991 Mark Andrews"
        }
    }
};
```

How Resource IDs are Assigned

By convention, resource ID numbers are divided into the following ranges.

Range	*Description*
-32768 through -16385	Reserved; do not use.
-16384 through -1	Used for system resources owned by other system resources (explained later).
0 through 127	Used for other system resources.
128 through 32767	Available for your use in whatever way you wish.

Resources With Special IDs

Some resources are subject to tighter restrictions than those in this list. For example, a device driver cannot have a resource ID greater than 31. Restrictions on numbers assigned to special kinds of resources can be found in the chapters dealing with those resources in *Inside Macintosh.*

Resource Names

Any resource may be given a resource name. The use of a resource name is usually optional since resources are usually referred by their resource IDs in application programs. But the resource specifications for some kinds of resources, for example, device drivers and desk accessories, always contain resource names.

A resource name, like a resource ID, should not be assigned to two resources of the same type; if the same name is given to two resources of the same type, the result of trying to access either one of them is unpredictable.

Resource names, unlike resource types, are case insensitive. When a resource name is passed to the Resource Manager, the Resource Manager ignores the case but does not ignore diacritical marks.

This is the source code for a typical 'TRAP' resource, a system resource that always contains a resource name:

```
resource 'TRAP' ($8A1,"FrameRect") {"QuickDraw", $A8A1};
```

Resource Attributes

If you want a resource to be treated in a certain way—for instance, to be treated as unpurgeable or to be loaded into memory as soon as your application is loaded—you can add one or more resource attributes to the resource's specification by changing the value of the resource description's attributes field. In the attributes field, each attribute of a resource is specified by a bit in the low-order byte of a word, as illustrated in Figure 6-2.

The Resource Manager provides a set of constants that you can use as resource attributes in resource specifications. The construction of a resource description's attributes field is shown in Listing 6-13.

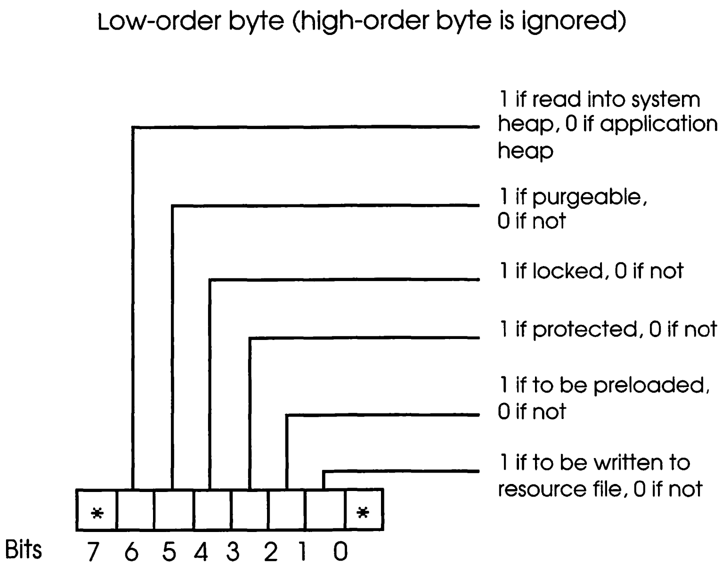


Figure 6-2. Resource Attributes

Listing 6-13. The resource attributes field

```
CONST  resSysHeap     = 64;   {set if read into system
                               heap}
       resPurgeable   = 32;   {set if purgeable}
       resLocked      = 16;   {set if locked}
       resProtected   = 8;    {set if protected}
       resPreload     = 4;    {set if to be preloaded}
       resChanged     = 2;    {set if to be written to
                               resource file}
```

Important ▶

Tips and Warnings about Resource Attributes. Here are some handy facts to know about when you are deciding how to set an application's resource attributes:

- You should not change the setting of bit 0 or bit 7 in the attributes field.
- You should not set the resChanged attribute directly; resChanged is set as a side effect of the Toolbox call Changed-Resource, which you call to tell the Resource Manager that you have changed a resource.
- You should also refrain from setting the resSysHeap attribute. If a resource with this attribute set is too large for the system heap, the bit is cleared, and the resource is read into the application heap.
- If the resProtected attribute of a resource is set, an application cannot use Resource Manager routines to change the ID number or name of the resource, modify its contents, or remove it from the resource file. However, you can make the Memory Manager call SetResAttrs, which sets the resource attributes, to remove the protection or just change some of the other attributes.
- A locked resource is neither relocatable nor purgeable, so the resLocked attribute overrides the resPurgeable attribute; when resLocked is set, the resource isn't purgeable regardless of whether resPurgeable is set.
- The resPreload attribute tells the Resource Manager to read a resource into memory immediately after opening the resource file. This procedure could be useful if, for example, you wanted to draw ten icons as soon as your application started. Instead of reading and drawing each one individually in turn, you can have all of them read into your application at once, and you could then just draw all ten.

How the Resource Manager Works

When the Macintosh system starts up, it initializes the Resource Manager and opens the system resource file. When an application starts up, its resource file is opened.

When an application needs to use a resource, it informs the Resource Manager. The Resource Manager first looks for the resource in the application's resource file. If other resource files have been opened—for instance, by other applications—the Resource Manager looks next in each of those files, normally beginning with the one most recently opened. (If you wish, you can change this order, as explained in the Resource Manager chapter of *Inside Macintosh.*)

If the Resource Manager cannot find the resource it is looking for in any application, it looks in the system resource file. Once it finds a requested resource, it reads the resource into memory and returns a handle to the calling program.

This system makes it easy for applications to share resources. It also means that you can override a system resource with a resource of your own. For example, you could override the system's alert-dialog resource to display a customized alert dialog of your own design.

The Resource Map and Resource Data

The resource fork of a file is divided into two main parts: a resource map and resource data. A resource map contains information that the Resource Manager can use to find any resource in the file. The data that follows the resource map is the resource data for each actual resource that the resource fork contains.

Note ▶

Just the Facts. The Resource Manager doesn't know or care about the individual formats of the resources that it manages. When you request a resource from the Resource Manager, it merely loads the resource into memory and then returns a handle to it. The program that has called the Resource Manager can then use the resource in any way it desires.

The Resource Map

The resource map in a resource fork is designed much like a header record in an ordinary disk file. The resource map contains information that enables the Macintosh system to find each individual resource in the stream of bits stored in the resource fork.

You'll probably never need to access a resource map directly; the job of interpreting resource maps and picking out the resources that they point to is the responsibility of the Resource Manager. But if you're curious about what the format of a resource map looks like, you can find it described and illustrated in the Resource Manager chapter of *Inside Macintosh*.

How Resource Maps Work

When a resource is stored on a disk, the resource map for the file in which it is stored contains an offset that points to the start of the resource's data, along with information that tells how long the resource is. If the resource has been read into memory, the resource map contains the resource's handle.

When a file is opened, its resource map is read into memory. The resource map stays in memory until the file is closed. However, individual resources can be removed from memory—either temporarily or permanently—by the Resource Manager or by the Memory Manager.

When an application requests a resource, the Resource Manager searches for it by looking through the application's resource map—not through the actual resources stored in the application's research fork. If the resource being requested has been placed in memory, the Resource Manager returns its handle to the calling program. If the resource has not been read into memory, the Resource Manager reads it into memory and then returns its handle.

Purgeable and Unpurgeable Resources

When you request a resource from the Resource Manager, you can instruct that it be read into memory as purgeable or unpurgeable data by giving it a resource attribute of "purgeable," as explained later in this chapter.

If you make a resource purgeable, the Memory Manager can purge it—that is, remove it from memory temporarily—when the space that it occupies is needed for other purposes. If you make a resource unpurgeable, it cannot be removed from memory unless you order its removal.

If the Memory Manager removes a purgeable resource from memory and you then want to use it again, the Resource Manager reads it back into memory and returns its new handle.

Since these processes all take place automatically, it is usually best to make your resources purgeable. That way, the Memory Manager can keep resources in memory only when they are needed and can perform its memory management tasks more efficiently.

However, if you create a small resource that is constantly in use—for example, a bit image that is used a cursor—you may want to make it unpurgeable. That will prevent the Memory Manager from slowing your program down by repeatedly purging your cursor from memory and then reading it back into memory from disk again.

With the notable exception of menus, all resources used by applications are read into memory as purgeable resources unless they are specifically defined as unpurgeable. Menus are stored as unpurgeable resources because they are used so frequently in Macintosh programs. They should never be made purgeable. (The source code for a menu resource was shown in Listing 6-6.)

Resource data is normally not read into memory until an application requests it. However, you can instruct the Resource Manager to place a resource in memory as soon as a resource file is opened by setting the "preload" attribute in its resource specification. In addition, the preload attribute of a menu resource is usually set, so that an application's menu will be read into memory as soon as the application is loaded.

How Resources Are Stored in Memory

When resource data is read into memory, it is stored in a relocatable block in the heap. That means that the Memory Manager can change the physical addresses of your resources at any time—even if you have made them unpurgeable. So when you work with resources, you must take care to lock, unlock, and dereference them properly in order to keep up with the operations of the Memory Manager. More information about working with the Memory Manager is presented in Chapter 7.

Once a resource has been read into memory, you can change its data in any way you like. However, your changes will not become permanent unless you request that they be made permanent and then either close or update your resource file.

Tools for Creating Resources

There are several ways to create and edit resources, and they all have their advantages and disadvantages. Tools that you can use to design and edit resources are described in the following sections.

ResEdit

An easy way to design and edit resources—particularly graphics-style resources, such as cursors and icons—is to use the ResEdit resource editor that is supplied with MPW and many other software development systems.

ResEdit is an interactive, graphics-based editor that is now equipped with a host of useful features, including a tool palette like those in graphics programs such as MacPaint and MacDraw. ResEdit underwent a complete facelift with the introduction of Version 2.1, so if you haven't used it in a while, you might want to check it out in its latest version and see what's been added.

A key feature of ResEdit is its extensibility. You can extend its capabilities in two ways: by creating editable templates for your own resource types and by writing your own resource picker, or editor, and adding it to the ResEdit system.

One improvement in ResEdit is that it now complies more closely with Apple's user interface guidelines. When you open ResEdit, you can now select a resource file using a standard file-selection dialog like the one shown in Figure 6-3. When you select a resource file to edit, ResEdit displays a file window in which each type of type is illustrated with an icon, as shown in Figure 6-4.

When you select a resource file to edit, ResEdit displays a menu bar like any other well-behaved Macintosh application. From the menu bar, you can choose from a wide variety of ResEdit operations.

ResEdit comes with a complete set of instructions, so procedures for using the package will not be described in detail here. But a few of ResEdit's newest features are worth pointing out.

The 'BNDL' Resource Demystified

When you're learning to program the Macintosh, one of the most mysterious resources you're likely to encounter is the bundle (or 'BNDL') resource, which provides the Finder with a package of resources used by an application. When you write an application, you're supposed to

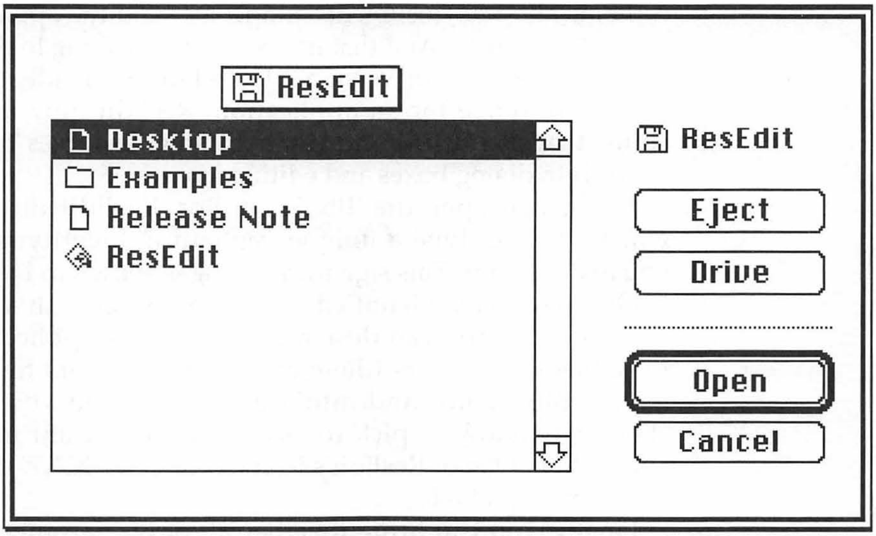


Figure 6-3. ResEdit open dialog

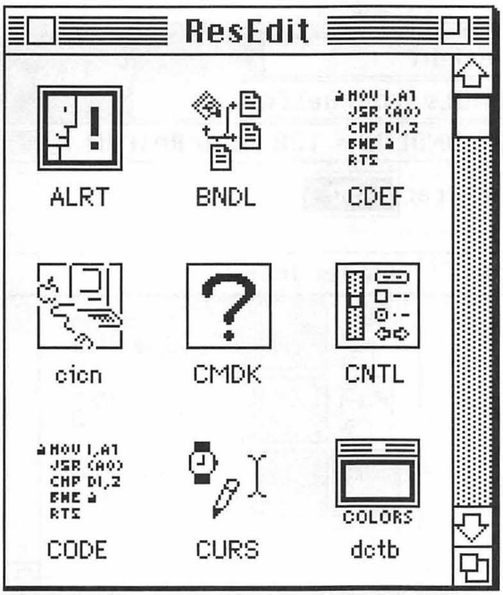


Figure 6-4. ResEdit icons

combine all of its documents and icons—and the application itself—into a 'BNDL' resource. And that has been a tricky thing to do, up to now.

With the release of Version 2.1, ResEdit has made it easier to create a 'BNDL' resource for an application. ResEdit now includes a 'BNDL' editor that you can use to create bundle resources by merely clicking controls in dialog boxes and editing icons.

When you open the 'BNDL' editor, ResEdit displays a dialog in which you can type a unique "signature" identifying the application you're designing. This signature dialog is shown in Figure 6-5.

Once you have identified your application with a signature, you're on your way. You can design icons for your application using one of ResEdit's icon editors (there are separate editors for black-and-white icons, color icons, and miniature icons), and you can use an Icon chooser window to pick the icons that you want to include in your application. One of ResEdit's icon editors, the 'ICN#' (icon list) editor, is shown in Figure 6-6.

Finally, you can bring together all of the resources in your bundle and examine all of them together in an "extended view" window.

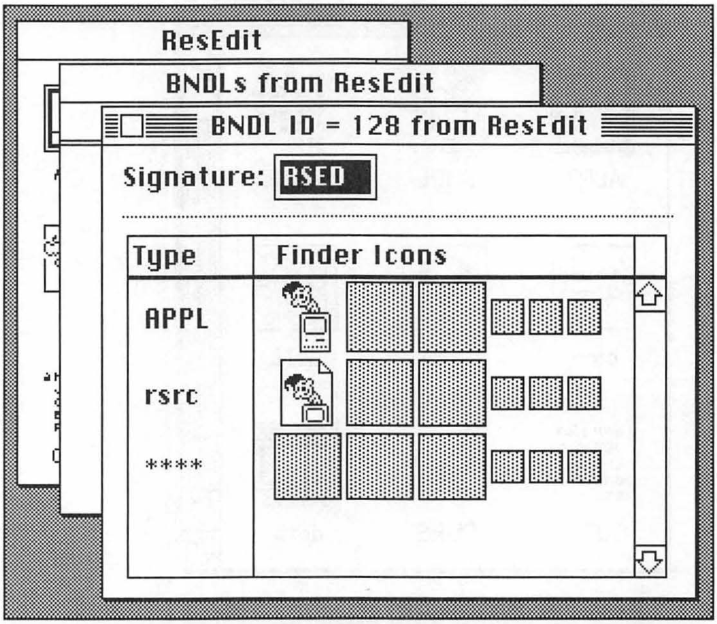


Figure 6-5. Bundle window

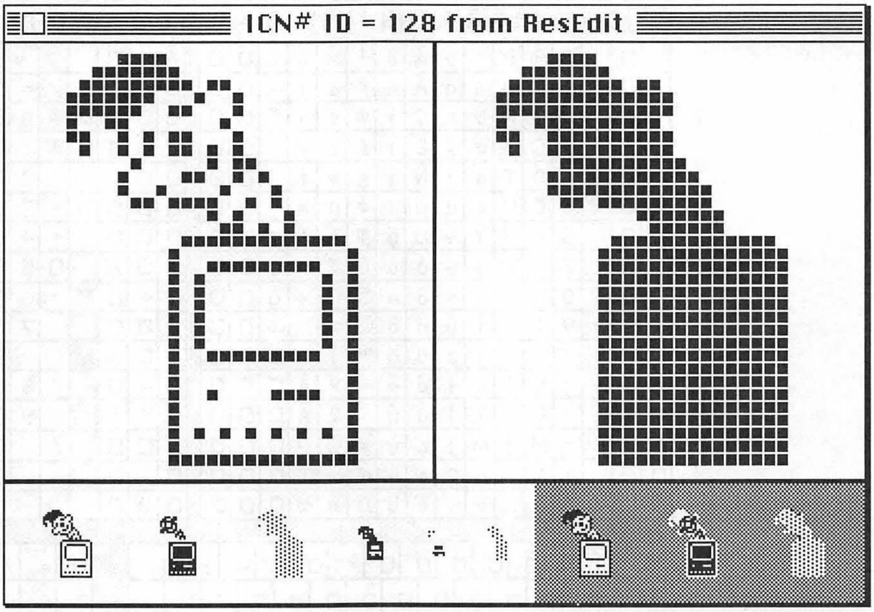


Figure 6-6. ICN# editor

One of the most welcome additions to ResEdit will certainly be the *Transform menu,* which lets you edit, or "transform," icons in a number of interesting ways. You can flip regions horizontal or vertically, rotate them, and even "nudge" them by a single pixel in any direction. You can also show or hide grid lines, or you can change the size of an icon for better editing.

▶ The 'KCHR' Resource

ResEdit has a very impressive tool for editing the 'KCHR' resource, which controls keyboard mapping. There is a window that shows a picture of a keyboard and all 256 characters in the currently selected font. You can display a character by clicking with the mouse in either the keyboard region or the virtual keycode. You can then assign a character to a key by dragging a character either onto a key on the keyboard or onto a character shown on the character chart.

The 'KCHR' editor has lots of special features, all explained in the ResEdit documentation. ResEdit's 'KCHR' editor is shown in Figure 6-7.

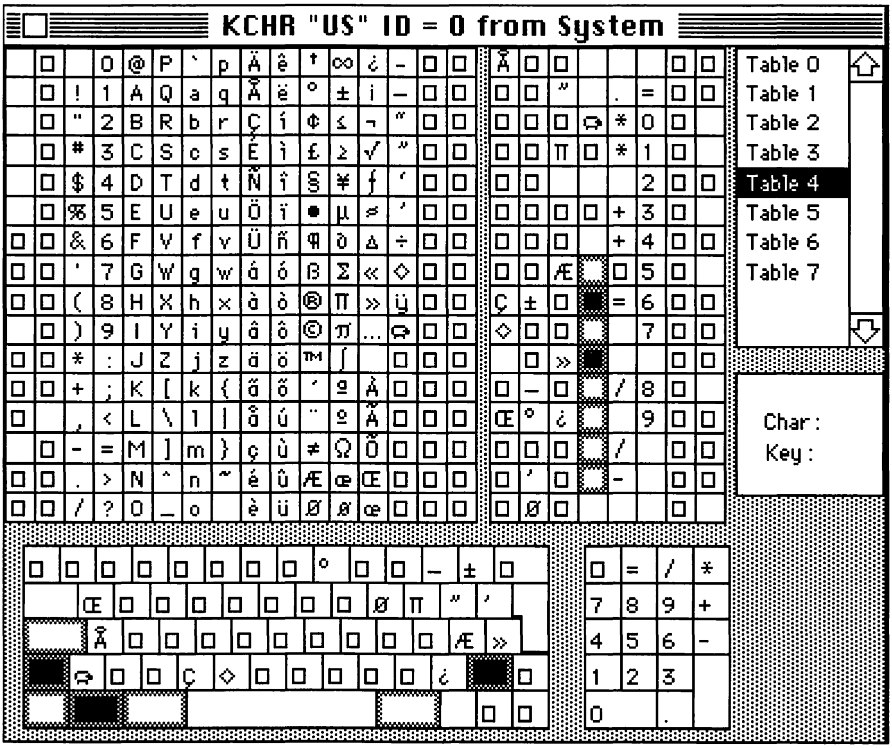


Figure 6-7. The 'KCHR' editor

Decompiling ResEdit's Resources

Although ResEdit is a resource designer's delight, it produces only resource data—no source code at all—for the resources it creates. That's not so bad for the casual programmer who just wants to knock out an occasional icon or dialog, but if you're into serious programming, you need resource definition files for the resources you design. Otherwise, there's no hard-copy record of what you've done and no way to automate the process, as you can with Rez.

▶ The MPW Editor

Fortunately, you can have your source code and use it too by combining the features of ResEdit with those of the MPW Editor. By using the MPW commands Rez and DeRez, as explained earlier in this chapter, you can design resources with ResEdit, and then decompile them into resource definition files with the DeRez command. Conversely, you can write resource definition files using the MPW Editor, compile them using Rez, and then edit them—or add new graphics resources—with ResEdit. And, at any time you like, you can recompile your resource definition files using the Rez command.

▶ SARez and SADeRez

You can also create resources using SARez and SADeRez: a pair of standalone applications that make MPW's Rez and DeRez tools available to programmers who don't have or use MPW. SARez and SADeRez work just like the Rez compiler and DeRez decompiler that come with MPW, but they can be used outside the MPW environment.

When you launch SARez, the standard Rez Commando dialog is displayed. You can then use the dialog to compile a resource description file into a working resource fork.

When DeRez is launched, it displays the standard DeRez Commando dialog. You can then use the dialog to decompile a resource fork into a resource description file written in Rez format.

SARez and SADeRez are shipped with THINK Pascal 3.0, and are available from Apple as standalone applications.

▶ Calling the Resource Manager

Once you have written and compiled a resource fork for an application, you can use the resources you have created by making calls to the Resource Manager. The Resource Manager is initialized automatically when you start up your Macintosh: The system resource file is opened and its resource map is read into memory. Then, when your application starts up, its resource file is opened.

Other useful Resource Manager calls are listed in Table 6-10.

Table 6-10. Resource Manager calls

Call	*Function*
GetResource	Returns a handle to a resource with a specified type and ID number, reading the resource into memory if desired.
GetNamedResource	Returns a handle to a resource with a specified type and name, reading the resource into memory if desired.

Table 6-10. Resource Manager calls (continued)

Call	*Function*
RmveResource	Removes the resource reference of a specifed resource in the current resource file. The resource data is not removed from the resource file until the file is updated.
ChangedResource	Makes changes that have been made to a specified resource permanent.
WriteResource	Writes the resource data for a specified resource to the resource file.
CurResFile	Returns the reference number of the current resource file.
CreateResFile	Creates a resource file.
OpenResFile	Opens the resource file having the given name and makes it the current resource file.
CloseResFile	Closes any resource file specified by reference number.
ResError	Reports any errors that may occur during execution of Resource Manager calls.
CountTypes	Returns the number of resource types in all open resource files.
GetIndType	Returns the types of specified resources.
CountResources	Returns the total number of resources of a specified type in all open resource files.
GetIndResource	Returns handles to resources of a specified type.
SetResLoad	Sets a flag that determines whether resources will be loaded.
LoadResource	Returns the handle of a resource and reads it into memory.
SizeResource	Reports how much memory space a resource will require.
UseResFile	Sets the current resource file to a specified file.
HomeResFile	Returns the reference number of a resource file containing a specified resource.
GetResInfo	Returns the ID number, type, and name of a specified resource.

Table 6-10. Resource Manager calls (continued)

Call	*Function*
GetResAttrs	Returns the resource attributes for a specified resource.
R	Reads a specified resource into memory.
AddResource	Adds resources to a resource file.
UniqueID	Returns an ID number greater than 0 that is not currently assigned to any resource of the given type in any open resource file. By using this number when you add a new resource to a resource file, you can ensure that you won't duplicate a resource ID.
DetachResource	Replaces the handle to a specified resource with NIL. The given handle will no longer be recognized as a handle to a resource; if the Resource Manager is subsequently called to get the detached resource, a new handle will be allocated.

The Creation.r file presented in Appendix D, is an example of a resource description file used in an actual application.

▶ Conclusion

In this chapter, you've seen how resources are used in Macintosh applications, and how resources are managed by the Resource Manager. You've also learned how to write a resource fork using Rez, DeRez, and ResEdit; how to check a resource fork for errors using RezDet and ResEqual; and how to manage resources in an application by making calls to the Resource Manager.

7 ▸ MPW and the Memory Manager

When an application won't work and you can't figure out why, the most likely reason is that something is wrong with the way that the program is using memory. The Macintosh has an extraordinarily elegant system for managing memory. When you write a Macintosh program, the computer keeps track of all physical memory locations for you, so you'll never have to hang memory maps on the wall and try to figure out where in memory to put this piece of code or that block of data. But you do have to understand how the system works. If you don't, you'll never be able to write a Macintosh program.

In a nutshell, the Macintosh manages memory with an operating system manager called the Memory Manager. When a well-behaved Macintosh application needs memory, it never just goes out and tries to find it; instead, it calls the Memory Manager. The Memory Manager finds and allocates the memory and then tells the application where it is. Conversely, when an application no longer needs a block of memory, it informs the Memory Manager. The Memory Manager then deallocates the memory, freeing it for other uses. Of course, to use this system, you have to know how to communicate with the Memory Manager.

In this chapter, we'll take a close look at how memory is laid out in the Macintosh, how the Memory Manager manages memory, and how you can use the Memory Manager to handle memory in programs written using MPW.

▶ Mapping the Macintosh

Figure 7-1 is a simplified memory map that shows how memory is laid out in a Macintosh computer when one application is running. No memory addresses are shown because different amounts of memory are provided in different models of the Macintosh, and extra memory can be installed in most models.

As you will see later in this chapter, the Macintosh memory map changes when more than one application is running under MultiFinder or under the System 7 Finder.

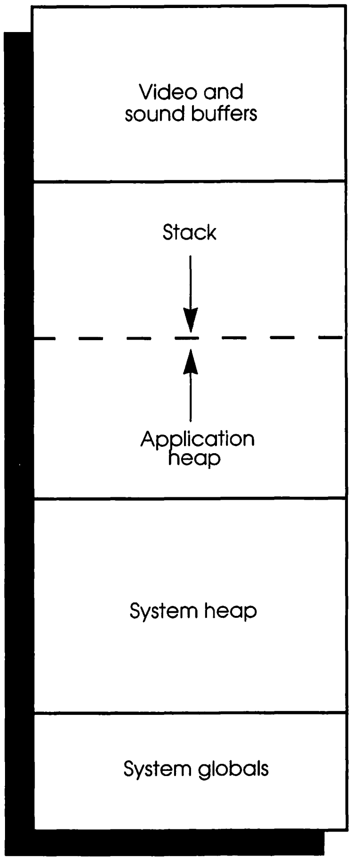


Figure 7-1. Simplified Macintosh memory map with one application running

As Figure 7-1 illustrates, the memory of a Macintosh is divided into five main blocks. Starting from low memory and moving upward, the five main sections of memory are as follows.

- System globals (beginning with memory address 0)
- The system heap
- The application heap (grows upward in memory)
- The stack (grows downward in memory)
- Buffers reserved for video, I/O, and sound

Although a Macintosh has five main memory blocks, an application can allocate and release memory dynamically in only two of those blocks. One of these areas is called the stack; the other is called the application heap, or simply the heap.

When an application allocates memory in its stack, the stack grows downward in memory, toward the application heap. When memory is allocated in the application heap, the heap grows upward in memory, toward the stack.

▶ Low-Memory Globals

The lowest area of memory, beginning at address 0, is used to hold system globals, sometimes referred to as low-memory globals. System globals, as their name implies, are global variables that are used by various parts of the Macintosh system. System globals can be used by applications as well as by the operating system. However, you should use system globals in your applications only when it is absolutely necessary; that is, when there is no Toolbox or operating system call that you can use to accomplish the same result.

Although the addresses and functions of system globals are published in *Inside Macintosh,* Volume III (Appendix D), Apple has never made any explicit guarantees that the system globals currently in use will remain the same in future models. However, Apple has guaranteed that applications won't become obsolete if they use approved Toolbox and operating system calls. So you should use Toolbox and operating system calls to obtain values whenever possible, and you should stay away as much as possible from using system globals.

There are hundreds of system globals, and they are used to hold all sorts of values. For example, system variable $014A (hexadecimal 14A) holds the address of the event queue, system variable $0824 holds the starting address of screen memory, and system variable $0A60 is used to store error codes returned by the Resource Manager. But you'll rarely, if ever, have to access any of these three variables. Generally speaking, you should use the Event Manager to manage events, call QuickDraw when you want to draw on the screen, and call ResErr to find out if the Resource Manager has returned an error.

▶ The System Heap

The system heap is the main area of memory used by the operating system. All system code that is executed while the Macintosh is running resides in the system heap. The system heap also contains various data structures used by the system.

Although system globals are stored in specific, documented addresses, the system heap is an area where code and larger data structures are stored dynamically. So the contents of the system heap change continually while the Macintosh is running. For example, the system heap contains a data structure called a Volume Control Block, or VCB, for every disk volume that's currently mounted. When you insert a disk in a disk drive, the system allocates a new VCB in the system heap and fills it in with information about the disk you've just inserted.

▶ The Application Heap

The application heap is an area of memory reserved for use by applications. When you launch an application, its code is stored in an application heap, along with all its resources such as menus, dialogs, and icons. In addition, objects that are created by a program—such as blocks of text and various kinds of data structures—are kept in the application heap. The memory used to hold these objects is allocated and deallocated through calls to the Memory Manager.

Note ▶

How You Can Use the System Heap. When you design a resource, you can specify that you want it placed in the system heap rather than in an application by setting its resSysHeap attribute, as explained in Chapter 6. The resource is then stored in the system heap when the application is launched and is available to any application.

When you write a program in assembly language, you can also place any data structure in the system heap rather than in an application heap by setting a field in the trap macros that call NewHandle and NewPtr. In Pascal or C, you can get a handle or a pointer to information in the system heap by making the call NewHandleSys or the call NewPtrSys.

The Application Heap and MultiFinder

Until the advent of MultiFinder, the Macintosh had only one application heap, which was used by the application currently running. But when you run a program under MultiFinder or under the System 7 Finder, every application on the desktop has its own heap. When the Macintosh user switches back and forth between applications on the desktop, the Finder keeps track of each application's "world": its application heap, its stack, and some system globals.

When you launch an application under MultiFinder or System 7, the system creates a new application heap and a new stack from available RAM. The size of the application is determined by two fields in its 'SIZE' resource (which was introduced in Chapter 6). When you design an application, you can specify its minimum size and its preferred size by setting two fields in its 'SIZE' resource: a "preferred size" field and a "minimum size" field. When the application is loaded into memory, it is allocated the amount of memory specified in the "preferred size" field of its 'SIZE' resource, if that is possible. Otherwise, the application is given the largest amount of memory available that is greater than or equal to the amount specified in the "minimum size" field. If that amount of memory isn't available, an error is returned.

When you exit an application being run in a MultiFinder environment, the memory occupied by the program's application heap is deallocated and becomes available for use by other applications.

MultiFinder and Low-Memory Globals

One tricky problem that MultiFinder faces when it switches from one application to another is what to do about system globals. The values of some system globals—for example, the contents of MenuList ($0A1C), which contains a handle to the application's current menu bar—differ from application to application, and the values can't just disappear into a black hole when an application temporarily moves from the foreground of the desktop into the background.

MultiFinder neatly solves this problem by making a separate copy of certain important low-memory globals for each active application. Then, when the user switches from application to application, the appropriate sets of globals are swapped into and out of memory, along with application heaps and stacks. Not all system globals are copied, however; MultiFinder is intelligent enough to know which globals are important enough to keep and which aren't.

The Stack

The stack is another area of memory in which an application can allocate and deallocate memory dynamically. When you declare a local variable in a program—for example, by using a VAR declaration inside a Pascal procedure or function, or by declaring a local variable inside a C function—the variable is placed on the stack. When the procedure or function ends, its local variables are pulled off the stack, and the stack space that they occupied is deallocated.

When an application uses global variables, they are placed just above the stack when the application is loaded into memory. They aren't removed from the stack until the user quits the application.

From the time an application is launched until the time it quits, the starting address of its global variables is kept in a 680X0 register called the A5 register, and also in a low-memory global called CurrentA5. Applications access their own global variables by checking the contents of either the A5 register or the system global CurrentA5.

Programs written in assembly language usually access their global variables via the 680X0 A5 register. Programs written in Pascal and C access their global variables the same way, but the programmer usually doesn't have to be aware of exactly how the job is done because the compiler that's used to write the program takes care of it.

More information on how global variables are accessed via the A5 register is presented later in this chapter.

How the Stack Works

Although the analogy isn't perfect, a computer stack is sometimes compared to a spring-loaded plate dispenser in a cafeteria, as shown in Figure 7-2. When you remove a plate from the top of the stack, the next plate in the stack becomes the top plate and moves up to replace the plate that has been removed. You can put plates on top of the stack at any time you like, but you can never get to the plates underneath the one that is currently on top until it is taken away.

In other words, a stack of plates is a LIFO (last-in, first-out) device: The last item that was pushed onto the stack is always the first to be pulled off the stack, and you can never get to the second item in the stack until the first one is removed.

Furthermore, the plates stored in a stack of plates are always *contiguous*; that is, there can never be an empty space between two of the plates. Unless a plate is on the top of the stack or on the bottom, there is always another plate above it and another plate below it.

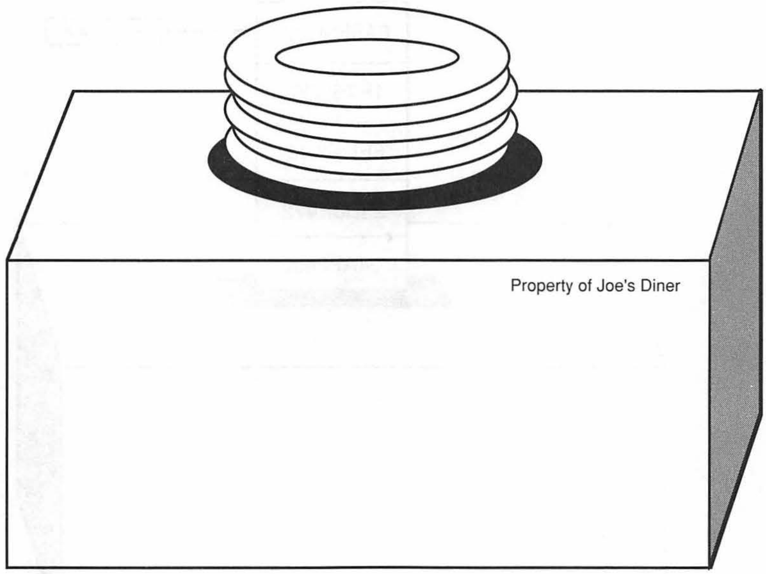


Figure 7-2. A stack

A stack in a computer is also a LIFO device. In a computer stack, just as in a stack of plates, the area of memory occupied by the stack is always contiguous. Since space is available only at the top of the stack—never in the middle—the stack can never contain any unallocated "holes." But an application heap *can* contain gaps, as you shall see later in this chapter.

The Stack Pointer

As noted earlier, comparing the stack with a stack of cafeteria plates isn't exactly accurate. Although plates can be physically removed from a stack, memory addresses are never actually removed from a computer. They always stay where they are, of course; but a stack pointer can be used to keep track of where the top of the stack is, as illustrated in Figures 7-3 and 7-4.

Figures 7-3 and 7-4 show a stack and a stack pointer. In Figure 7-3, the stack pointer is pointing to a memory location holding the value 6A5B9C32, at the top of the stack.

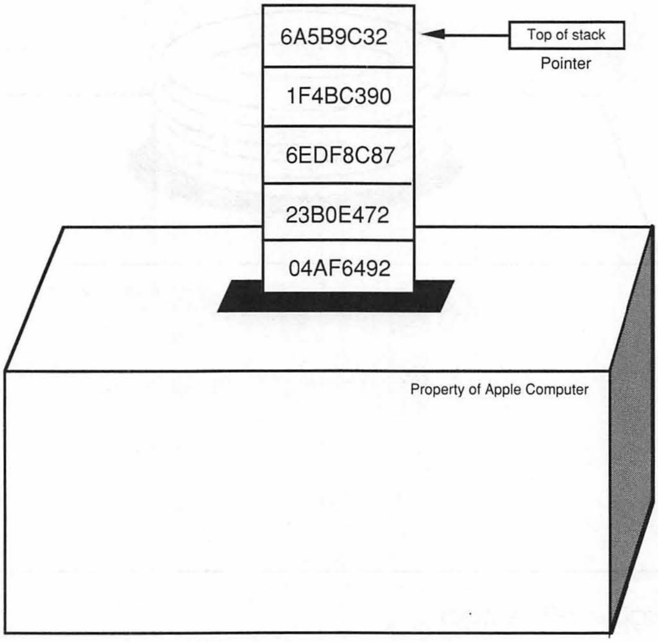


Figure 7-3. A stack with a pointer

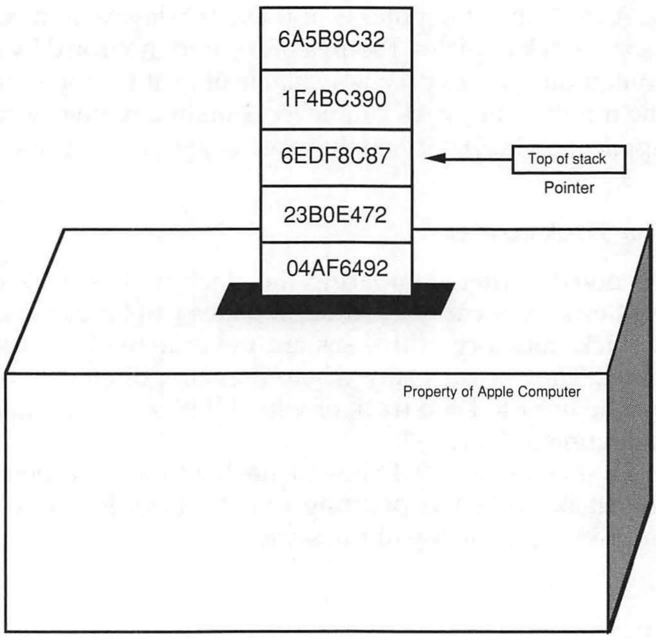


Figure 7-4. Moving the stack pointer

In Figure 7-4, the values 6A5B9C32 and 1F4BC390 have been "removed" from the stack. Physically they are still there, but the pointer now points to the memory address holding the value 6EDF8C87. This means that the memory address that holds the value 6EDF8C87 is now considered to be at the top of the stack.

Thus, when a piece of information is "removed" from the stack, the pointer is changed to point to a location closer to the bottom of the stack; and when a piece of information is placed on the stack, the pointer is changed to point to the new top of the stack. So the pointer always points to the memory address that is considered the top of the stack.

In the 680X0 microprocessor, as in Figures 7-3 and 7-4, a register called the stack pointer is used to keep track of information stored on the stack. However, unlike the stack pointer shown in Figures 7-3 and 7-4, the stack pointer in the 680X0 always points to the *next available stack location.* Each time a piece of data is pushed onto the stack, the value of the stack pointer is incremented; and each time a piece of data is pulled off the stack, the value of the stack pointer is decremented. So the stack pointer always holds the address of the *next available memory location* in the block of Macintosh memory used as a stack.

Another difference between a real computer stack and the stacks shown in Figures 7-3 and 7-4 is that a computer stack usually grows from higher memory address toward lower memory addresses. In other words, the *bottom* of a computer stack is at a *higher* memory address than the *top* of the stack. Therefore, Figure 7-5 is a more accurate illustration of how a stack really works in the Macintosh.

In Figure 7-5, the stack shown in earlier figures has been turned upside down. Also, memory addresses have been added to the illustration to show you that the bottom of the stack is actually at a higher memory address than the top of the stack. Notice that the memory addresses shown in the illustration progress in increments of two; that's because each address on a stack must be large enough to hold a 16-bit word.

Finally, in Figure 7-5, the stack pointer holds the address 27A4FE: the *next available memory address* on the stack. That's the way a stack really works; the next value pushed onto the stack shown in Figure 7-5 will be placed in memory address 27A4FE, and the stack pointer will be *decremented* to point to memory address 27A4FCD.

Suppose that the values 6A5B9C32 and 1F4BC390 are removed from the stack shown in Figure 7-5. What memory address does the stack pointer hold now? If your answer was 27A502, you're right; the stack pointer always holds the address of the next available memory location on the stack.

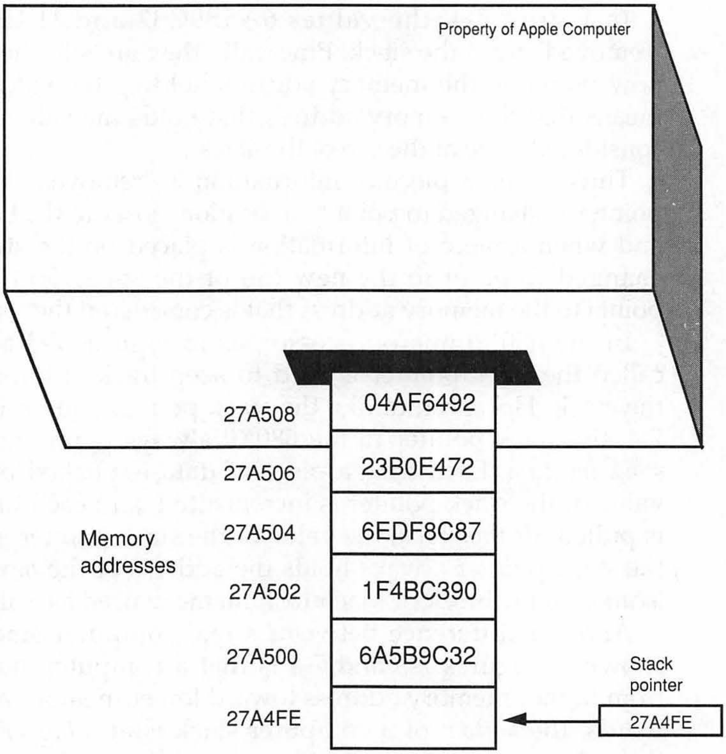


Figure 7-5. How a stack really works

Avoiding Stack Corruption

When you write a program in assembly language, every routine that you write must leave the stack in exactly the same state in which it was found. In other words, when a routine ends, the stack pointer must always have the same value that it had when the routine began. The reason that this is so important in assembly language is that, in programs written in assembly language, all routines share the use of the stack. If another routine has pushed a value onto the stack before your routine begins, and your routine goes looking for the value in the same place after your routine ends, the value had better be there; if it isn't, a system crash is almost inevitable.

▶ Pointers, Handles, and Heaps

A stack is designed to hold relatively small amounts of information for relatively short periods of time—usually, local variables. When you want to store longer blocks of information or you want to keep the information around for a longer time, the place to keep the information is in the application heap.

When an application needs to allocate or deallocate memory in its heap, it calls the Memory Manager. The Memory Manager then does all the necessary housekeeping to keep track of memory as it is allocated and released. Since an application can request almost any amount of memory from the Memory Manager and can request it at almost any time, a heap doesn't grow and shrink in an orderly way like a stack. In fact, after a program has been running for a while, its heap tends to become fragmented into a patchwork of allocated and free blocks, as shown in Figure 7-6.

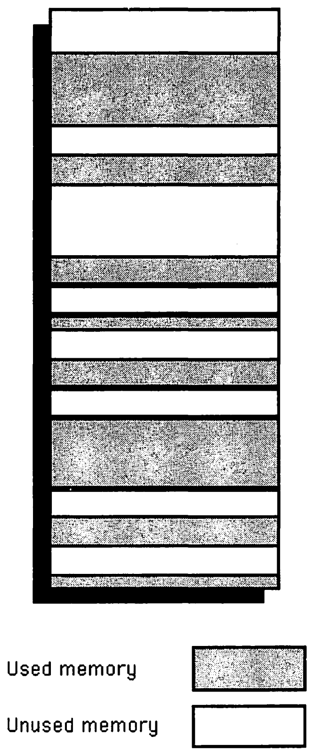


Figure 7-6. A fragmented heap

By the Way ▶

A Contiguous Observation. According to Apple expert Scott Knaster, the editor of the Macintosh Inside Out series, the stack is a LIFO structure and the heap is an LIOF structure. LIOF, he says, stands for "Last in, OK, fine."

When a heap has become fragmented, and a program asks the Memory Manager to allocate a new block of a certain size, the Memory Manager may not be able to satisfy the request even if enough free space is available. That's because the memory space that is available in the heap may be broken up into blocks smaller than the requested size. When this kind of situation occurs, the Memory Manager tries to create enough space to satisfy the application's request by compacting the heap, that is, by moving blocks of allocated memory together in order to coalesce the available space into a single larger block, as shown in Figure 7-7.

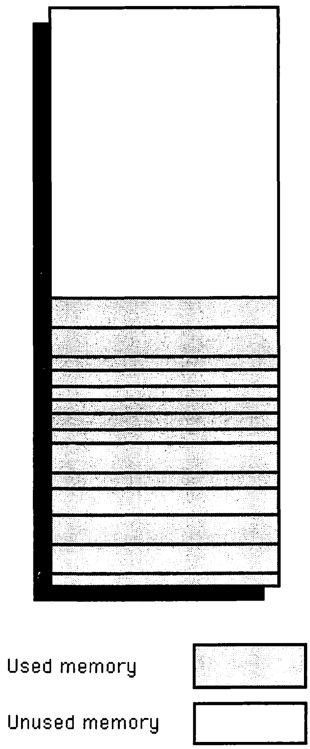


Figure 7-7. A compacted heap

▶ Pointers and Handles

When you request a block of memory from the Memory Manager, you can specify that it be made relocatable or nonrelocatable. Relocatable blocks are blocks of memory that the Memory Manager can move during heap compaction. Nonrelocatable blocks are blocks of memory that the Memory Manager will never move.

Nonrelocatable Memory Blocks

If you ask for a nonrelocatable block of memory, the Memory Manager allocates the block and returns a pointer to it. You can then access the memory by simply using the pointer that the Memory Manager has assigned.

Once a block of nonrelocatable memory has been allocated, it stays in its initial location until you remove it from memory; as long as it remains in memory, the Memory Manager will never move it in an effort to defragment the heap.

Although it's easy to access a nonrelocatable memory block—you can merely refer to it by its pointer—you should try to avoid using nonrelocatable blocks in Macintosh programs. If you put a lot of nonrelocatable blocks in an application, memory can become hopelessly fragmented.

You can request a block of nonrelocatable memory by using the Memory Manager call NewPtr. In Pascal, the syntax of the NewPtr call is

```
FUNCTION NewPtr (logicalSize: Size) : Ptr;
```

where Size is a data structure defined in the Memory Manager's interface file as a long integer. In C, the calling sequence of the NewPtr call is

```
Ptr NewPtr(Size logicalSize);
```

In a program written in Pascal, you could use the NewPtr call like this:

```
myPtr := NewPtr(anySize);
```

In a C program, you could do it this way:

```
myPtr = NewPtr(anySize);
```

When you're finished with a block of nonrelocatable memory, you can free it for other uses and dispose of its pointer by making the Memory Manager call DisposPtr (yes, that's the way it's spelled). In Pascal, the syntax of the DisposPtr call is

```
PROCEDURE DisposPtr (p: Ptr);
```

In C, this is the format:

```
void DisposPtr(Ptr p);
```

Pointers, Handles, and Relocatable Blocks

If you ask the Memory Manager for a *relocatable* block of memory, the Memory Manager allocates the block, assigns a pointer to it, and places that pointer in a table of master pointers stored in a nonrelocatable memory block in the heap.

The Memory Manager then returns a handle: the address of the master pointer that has been assigned to the memory block that you requested. From that point on, you can access the block of memory that you have requested by using its handle.

If you've never worked with handles, you might find the terms "pointer," "master pointer," and "handle" a little confusing. Remember that the *value* of a handle is the *address* of a master pointer; the *value* of a master pointer is the *address* of a relocatable block of memory.

Figure 7-8 illustrates the way this all works. The illustrations show a block of memory that the Memory Manager has moved. In the second illustration, the contents of the master pointer have been changed to point to the block's new location, but the location of the master pointer remains the same. Thus the handle, which points to the master pointer, can still be used to access the moved memory block.

Why You Should Use Relocatable Blocks

The use of relocatable memory blocks is highly recommended in Macintosh programs, since the Memory Manager can move relocatable blocks around at will whenever it needs to compact the stack and free larger sections of memory.

When the Memory Manager moves a relocatable block to a different memory location, it removes the block's old pointer from the table of master pointers kept in the heap, and replaces it with the block's new pointer. But the block's handle—now the pointer to its new master

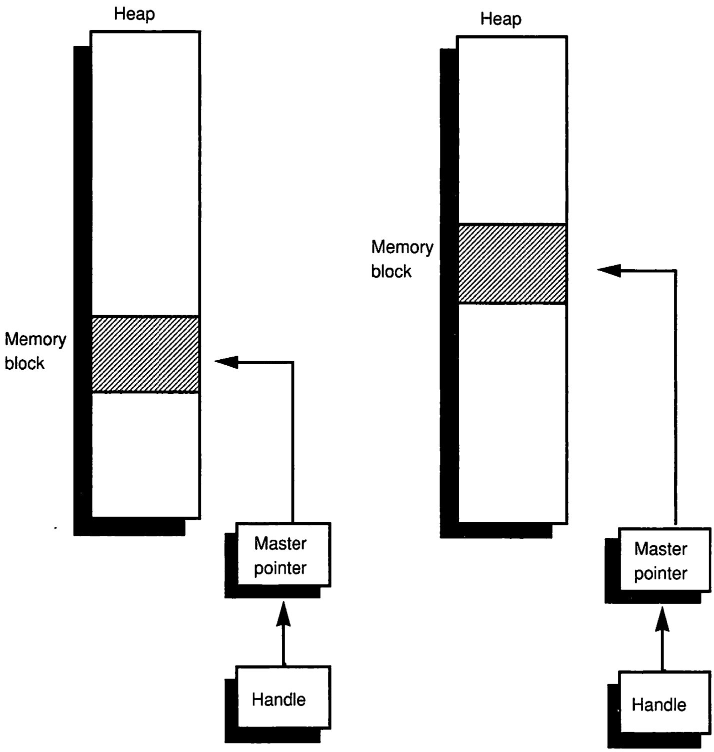


Figure 7-8. How master pointers and handles work

pointer—remains the same. So, no matter how many times a Memory Manager moves a relocatable block, you can always access it by using its handle.

The Structure of a Master Pointer

Figure 7-9 shows how a master pointer is configured in a Macintosh system that uses 24-bit addressing. Prior to the introduction of Software System Version 7.0 and the Macintosh IIci, master pointers were always configured as illustrated in Figure 7-9. With the introduction of System 7, however, some models of the Macintosh now support 32-bit addressing and the structure of a master pointer is different.

In a 24-bit addressing system as well as a 32-bit addressing system, a master pointer is a long word; that is, a word that is 32 bits long. When 24-bit addressing is used, the low-order three bytes of the word contain

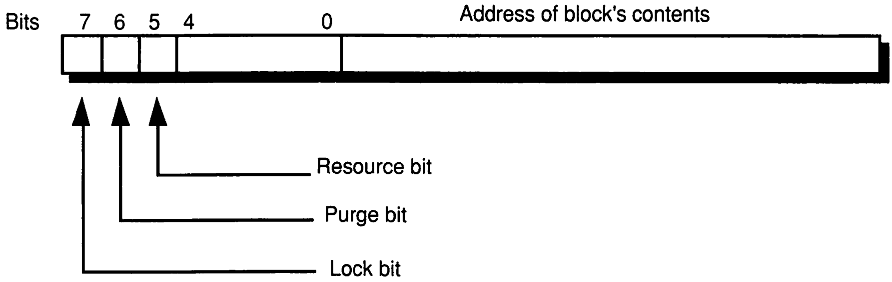


Figure 7-9. Structure of a master pointer

the address of the block's contents, and the high-order byte contains flag bits that specify the block's current status.

When 32-bit addressing is used, a master pointer contains a 32-bit address and therefore has no bits left over to be used as flags. Consequently, in a hardware and software configuration in which 32-bit addressing is used, the operating system stores information about master pointer flags in data structures set aside for that purpose rather than storing it within the 32-bit word used for the pointer itself.

Note that applications do not have to be aware of the structure of a master pointer if they follow Apple's developer guidelines and use routines provided by the Memory Manager for setting and clearing master pointer flags. For instance, to set or clear a master pointer's *Lock* flag, applications should use the HLock and HUnlock routines instead of directly accessing the flags in the master pointer.

If an application sets and clears master pointer flags directly rather than using Memory Manager calls that achieve the same results, the application will not execute correctly in environments that support 32-bit addressing. More information about 32-bit addressing is provided later in this chapter.

Master Pointer Flags

When 24-bit addressing is used, this is the structure of the high-order byte of a master pointer:

- Bit 7 is called the lock bit. It is set to 1 if the block is locked, and cleared to 0 if the block is unlocked.
- Bit 6 is the purge bit. It is set to 1 if the block is purgeable, and cleared to 0 if it's unpurgeable.

- Bit 5 is used by the Resource Manager to identify blocks containing resource information; in this bit, such blocks are marked by a 1; blocks without a resource are marked by 0.

Note that the flag bits in the high-order byte have numerical significance in any operation performed on a master pointer. For example, the lock bit is also the sign bit.

▶ Blocks That Are Always Nonrelocatable

Now that you know why you should avoid the use of nonrelocatable blocks, you're ready to hear the bad news. Some kinds of memory blocks are always nonrelocatable. They are listed here.

- *Master pointer blocks*: Master pointer blocks are blocks of memory that hold the master pointers to blocks of memory that *are* relocatable. If a master pointer block could be moved around in memory, the handles that are used to access master pointers wouldn't have anything permanent to point to, and the Macintosh wouldn't work at all. You can, however, prevent an application's master pointer blocks from fragmenting memory by placing all the master pointer blocks that you'll be using at the bottom of the heap as soon as your application is launched. Procedures for doing this are outlined later in this chapter.
- *GrafPorts*: A GrafPort, as explained in the "QuickDraw" chapters in *Inside Macintosh,* is a data structure that QuickDraw uses for drawing operations. When you want to access a GrafPort in a program, you must use a pointer rather than a handle. To prevent GrafPorts from fragmenting an application's heap, you should initialize QuickDraw and set up any GrafPorts you'll be using as soon as possible after you launch your program, as explained later in this chapter.
- *Window records:* A window record is a nonrelocatable object because it contains a GrafPort. To keep window records from fragmenting your application's heap, you should decide how many windows your program will need and set them up as early as possible in your program. Also, it helps to assign all your window records from the main segment of your program. Procedures for segmenting programs are also described later in this chapter.
- *Dialog records:* A dialog record contains a window record, so dialog records are also nonrelocatable. Dialog records aren't as likely to cause long-term heap-fragmentation problems as other kinds of

windows are, because a dialog window is typically opened, used, and disposed of rather quickly. Nevertheless it's still a good idea to assign dialog records early in an application, and only from the application's main memory segment.

Important ▶

What's the Hurry? Why should nonrelocatable blocks be allocated as soon as a program is launched? They should be allocated that soon because the Memory Manager is smart enough to look for memory at the bottom of the heap when it is available. The sooner you ask the Memory Manager for a block of nonrelocatable memory, the more likely the Memory Manager will be to place the block as low on the heap as possible. Since the block stays where it is for the duration of your application, it won't cause the heap to become fragmented during compacting operations.

The NewHandle Call

To request a block of relocatable memory, you can use the Memory Manager call NewHandle. In Pascal, the syntax is:

```
FUNCTION NewHandle (logicalSize: Size) : Handle;
```

where Size is a data structure defined in the Memory Manager's interface file as a long integer. In C, the calling sequence of the NewHandle call is:

```
Handle NewHandle(Size logicalSize);
```

In a Pascal program, you could write a NewHandle call this way:

```
myHandle := NewHandle(someSize);
```

In a program written in C, you could write it like this:

```
myHandle = NewHandle(someSize);
```

When you're finished with a block of relocatable memory, you can free it and discard its handle by making the Memory Manager call DisposHandle. In Pascal, the syntax of the DisposHandle call is

```
PROCEDURE DisposHandle (h: Handle);
```

In C, this is the format:

```
void DisposHandle(Handle H);
```

Dangling Pointers, and How to Avoid Them

Although you should use relocatable memory blocks whenever you can, there's one problem that the use of relocatable blocks can cause—and it's a common cause of bugs in Macintosh programs.

The problem can be illustrated with this scenario:

Suppose you have created a relocatable block by calling NewHandle, and you want to refer to it using a pointer. You might want to do this, for example, if you're repeatedly accessing the structure inside a loop: Access with a pointer would save a dereference each time through the loop, thus speeding it up. So, you dereference the handle to get a pointer, then tuck the pointer away for safekeeping—say, in a local variable.

Next, your program makes a call to some other manager. This call requires an allocation of some memory, but the Memory Manager can't find enough memory. So it compacts the heap. While this is going on, the structure that you plan to access using your pointer is moved somewhere else in memory.

Now you try to access your information by using the pointer that you have stored in a local variable.

Guess what; it isn't there. It's somewhere else now. So your program crashes. You have fallen victim to something commonly known as a dangling pointer: a pointer that points nowhere in particular, because the information that it used to point to has been moved to another location in memory.

The NewHandle call returns a handle, rather than a pointer. So, when you want to access a block of memory that you have allocated using NewHandle, you must *dereference* the block's handle. When you dereference the handle, you have a pointer.

This is the format for dereferencing a handle in Pascal:

```
myPtr := myHandle^;
```

In C, the equivalent is:

```
myPtr = *myHandle;
```

Another Way to Create a Dangling Pointer

Here's another way that you can wind up with an invalid, dangling pointer:

1. You allocate a block of memory, and get a handle to it, by using the NewHandle call.
2. By using dereferencing, you obtain a pointer that you can use to access your relocatable block. Then you store your pointer in a variable.
3. Before you get a chance to use your pointer, a call is made that causes the Memory Manager to compact memory.
4. Then you use the pointer that you have obtained by dereferencing—but the block of memory that the pointer pointed to after Step 2 has now been moved!
5. Result: A dangling pointer, and a potential crash in your program.

It's very important to remember this scenario, because it's a common cause of crashes in Macintosh programs. Heap compaction can only occur as the result of a few operating system calls, all of them in the Memory Manager: NewHandle and NewPtr are the most commonly used ones. But there's a complication: other Toolbox and operating system calls may, during their execution, call upon one of the compaction-causing routines, and this isn't always easy to predict. For example, the seemingly-innocent HideWindow call can cause compaction, because it may allocate memory for QuickDraw regions.

Apple publishes a list that indicates which Toolbox and operating system calls may cause compaction, but this list changes with every new system software release, so it's difficult to have up-to-date information. The safest course of action is to assume that any Toolbox or operating system call may cause compaction.

Another dangerous practice is to obtain a pointer by dereferencing a handle, and then to call a routine *in your own program* before you use the pointer you have obtained. The routine that you call may look safe enough, but it may be in another program segment. If it is in another segment, and if the segment has to be loaded into memory by the Segment Loader, that can cause memory compaction.

Finally, danger may be lurking in your compiler. Some Pascal or C library routines, such as printf, can make unannounced Toolbox or operating system calls that result in memory relocation.

The HLock and HUnlock Calls

There are two Memory Manager calls that can help you make sure that no dangling pointers creep into an application. One is HLock, which locks the block of memory associated with a handle, so that the block can't be moved. The other call is HUnlock, which frees a locked block of memory. You must be careful with these calls, though, because they can cause heap fragmentation.

You can call HLock just before (or immediately after) you dereference a handle. The block of memory associated with the handle then becomes temporarily unrelocatable. Once you have locked a relocatable block using HLock, the Memory Manager refuses to move it until it is unlocked with the HUnlock call. So you can prevent the creation of a dangling pointer by taking these steps:

1. Allocate a block of memory using NewHandle.
2. Obtain a pointer to the block by using dereferencing.
3. Lock the handle using HLock.
4. Use your pointer in any way you like; it's safe now.
5. When you have finished using your pointer, unlock the block which it accesses by using HUnlock.
6. If you need to access the same block of memory later on in a program, follow the above Steps 1 through 5 again.

Using the HLock and HUnlock Calls

The HLock call is very easy to use; in Pascal, its syntax is:

```
PROCEDURE HLock (h: Handle);
```

In C, the format is:

```
void HLock(Handle h);
```

The syntax of the HUnlock call is the same. In Pascal:

```
PROCEDURE HUnlock (h: Handle);
```

And in C:

```
void HUnlock(Handle h);
```

Once you know how to use HLock and HUnlock, you have no excuse for writing a program that contains a dangling pointer; just lock every handle you dereference just before, or just after, you dereference it. But when you're finished with a handle that you have dereferenced and locked, be sure to restore it to its original state by unlocking it.

Most important, make sure that your program doesn't cause any memory to be allocated while a handle is locked. If you do that, or if you forget to unlock a handle that has been locked, you can fragment your computer's memory.

Double-Dereferencing

Although the calls HLock and HUnlock are a sure-fire defense against dangling pointers, it's not absolutely necessary to use them every time you dereference a handle—as long as you're very, very careful.

When you write a program in C or Pascal, there's another technique for avoiding dangling pointers that's called double-dereferencing. Double-dereferencing is even safer than using HLock and HUnlock, and it's easier—because your compiler does it for you.

Using Double-Dereferencing

In Pascal, you can double-dereference a handle in much the same way that you get the contents of a pointer: by using the special character ^, but typing it twice. Suppose, for example, that you had a data structure stored in a relocatable block of memory, and that the handle to the structure was named myStructure.

In a Pascal program, you could obtain the value of any field in the data structure in one step, by writing a line like this:

```
valueOfField := myStructure^^.field1;
```

In C, it's the same story: use the pointer-access symbol *, but use it twice. In a C program, you could double-dereference a handle in a single step using this construction:

```
valueOfField = (**myStructure).field1;
```

The process is easy: Just use the handle in the same way that you would use a pointer, but use two pointer-access symbols instead of one. When you double-dereference a handle in this fashion, there is no need to call HLock or HUnlock, since the whole process can be written in a single line of code.

Macintosh lore is full of stories about how dangling pointers can cause programs to self destruct. But if you use pointers and handles carefully, you need never fall victim of such a catastrophe.

▶ Using the Memory Manager

You don't have to initialize the Memory Manager in order to use it. The Memory Manager is automatically initialized at startup time, and a system heap zone is automatically allocated.

A Macintosh heap is made up of three kinds of heap blocks: relocatable blocks, nonrelocatable blocks, and free blocks. Free blocks are blocks that might once have been allocated, but no longer are.

Every heap begins with a heap zone header, which provides important information about the heap to which it's attached. Fields in the heap zone header specify such things as the number of free bytes in the zone, the block of memory that's next in line to be purged, and the number of master pointers that have been allocated to the block with the MoreMasters call (described later in this chapter). The first byte of data in the heap is the last byte of the heap zone header.

Each block in a heap starts with a block header, which contains vital information about the block, including a field that tells which of the three varieties of heap blocks it is. The contents of the block—the area where the block's actual data is stored—follows the block header.

The only time you're likely to access a heap zone header or a block header directly is when you use MacsBug to debug a program. If you're interested in finding out more about such esoteric topics as heap zone headers and block headers, see the "Memory Manager" chapters of *Inside Macintosh.*

▶ How the Memory Manager Allocates Space

Each time an application is launched, an application heap and stack are initialized. By default, an application's stack is allocated 8K of memory, and its heap is allocated 6K. In addition, an area of memory called a growable heap space is set aside in case more stack or heap space is needed.

This area designated as growable heap space is situated between the stack and the heap. It lies in the area that contains the dotted line in Figure 7-1. It can be used by either the stack or the heap, whichever asks for it first. However, it is called growable *heap* space because it is usually claimed by the heap; most applications need much more memory space for their heaps than for their stacks.

How Heap and Stack Space Are Allocated

The initial allocation of an application's memory is determined by the values of several system globals.

The size initially allotted to the stack comes from a global variable named DefltStack, at memory address $0322 (hexadecimal 322). A pointer to the start of the heap is kept in a global variable called ApplZone, at address $02AA. A global variable named HeapEnd, at address $0114, holds the address of the end of the heap. Yet another global variable—ApplLimit, at address $0130—contains a pointer to the end of the growable heap space region.

▶ Master Pointer Blocks

When an application starts up, it is also allocated a master pointer block, in which all of its master pointers will be stored. Initially, this block is only large enough to store 64 master pointers—usually not nearly enough for a medium- to large-sized application. If you think your application might need more pointers than that—and it's better to err on the side of too many than too few—you can get more master pointer blocks by making the Memory Manager call MoreMasters.

The syntax of the MoreMasters call is simple; it takes no parameters. Just execute the statement

```
MoreMasters;
```

(using a "for" loop, if you like) for each additional block of 64 master pointers that you think you'll need.

How many additional master pointer blocks *will* you need? It's a good idea to allocate at least three and to use more (perhaps considerably more) if you're writing a large program. A master pointer block requires only 264 bytes of memory—4 bytes for each master pointer, plus 8 bytes for a header—so the use of MoreMasters isn't very costly.

▶ Tips on Memory Management

When your application starts up, it should allocate the memory it requires in the most space-efficient manner possible, arranging things in such a way that most of the nonrelocatable blocks it will need are stored together at the bottom of its heap. One call that you should make as soon as possible is the Memory Manager procedure MaxApplZone, which expands the application heap zone to its limit. You can then call

MoreMasters several times to allocate as many blocks of master pointers as you think your application will need.

That done, you should initialize QuickDraw and (if your application uses windows) the Window Manager. Finally, you should allocate space for any GrafPorts, window records, and dialog records that your program will be using. If you do that quickly enough, the memory manager places all your GrafPorts, window records, and dialog records at the bottom of the heap, and those three kinds of structures—which are always stored as nonrelocatable objects—won't cause any heap-fragmentation problems later on.

▶ QuickDraw Globals

When QuickDraw is initialized by an application written in MPW, a portion of the application's stack is reserved for a set of global variables that are used in QuickDraw operations. These variables are known, logically enough, as QuickDraw globals. When an MPW application initializes QuickDraw, the QuickDraw globals are all placed on the application's stack, just below the application's own global variables. That means that QuickDraw globals are placed on the stack *after* the application's own global variables have been allocated.

There are more than 200 QuickDraw globals, but only nine of them are directly accessible from application programs. Those nine globals are listed in Table 7-1. As the table shows, the first variable in QuickDraw's list of global variables is a pointer called thePort. This pointer, a variable defined as thePort in QuickDraw's interface files, points to a GrafPort. Other public and private QuickDraw globals are used to set up thePort's drawing environment: patterns, font data, and so on.

Table 7-1. QuickDraw globals

Variable	*Type*	*Offset from thePort*
qd.thePort	GrafPtr	0
qd.white	Pattern	-8
qd.black	Pattern	-16
qd.gray	Pattern	-24
qd.ltGray	Pattern	-32
qd.dkGray	Pattern	-40
qd.arrow	Cursor	-108
qd.screenBits	Bitmap	-122
qd.randSeed	Long	-126

One important QuickDraw global is the one identified as screenBits in Table 7-1. The screenBits variable is a data structure that defines the bitmap in which QuickDraw does its drawing. Normally, screenBits defines QuickDraw's drawing area as the screen.

In programs written under MPW, you can access QuickDraw's globals by using the qd constant. This constant is also defined in QuickDraw's interface files, and it can be treated as a data structure. For example, you can access the QuickDraw global screenBits by using the qd.screenBits.bounds constant.

More information about QuickDraw, and how QuickDraw's global variables work, can be found in the "QuickDraw," "Color QuickDraw," and "Assembly Language" chapters of *Inside Macintosh*.

By the Way ▶

How QuickDraw Globals Stack Up. QuickDraw is the only Toolbox manager whose global variables are placed on the application stack. All other Toolbox and operating system managers keep their variables in low memory instead of on the stack.

Since QuickDraw globals are allocated after an application's variables are allocated, the starting address of QuickDraw's list of globals may vary. Since the globals are placed in memory by being pushed onto a stack, the GrafPort pointer called thePort is the QuickDraw global with the *highest* memory address.

▶ The A5 World

To keep track of where an application's QuickDraw globals start, the operating system places their starting address in a 680X0 register called the A5 register. Since the first QuickDraw global immediately follows the last application global on the stack, you can use the A5 register to access an application's own global variables as well as to access its QuickDraw variables. By applying a negative offset to the contents of the A5 register, you can access any QuickDraw global. By applying additional negative offsets, you can access the application's own global variables since they are placed on the stack *before* the application's QuickDraw globals are allocated.

The Macintosh operating system also uses the contents of the A5 register to access a jump table: a table that allows routines in one segment of a program to call routines in another segment (the segmentation of programs is explained later in this chapter). The jump table used by an application always starts 32 bytes above the address contained in the A5 register. Thus jump table addresses can be calculated as positive offsets to the contents of A5.

Since the A5 register can be used to calculate so many different addresses, the addresses that can be accessed from the contents of the A5 register are often referred to as an application's A5 world. An application's A5 world includes its global variables, its QuickDraw globals, and its jump table addresses. Negative offsets to the contents of A5 refer to QuickDraw globals and application globals, whereas positive offsets refer to the contents of the jump table. Register A7, the stack pointer, holds the address of the top of the stack.

▶ Initializing QuickDraw

To initialize QuickDraw, you must make the QuickDraw call InitGraf. In Pascal, InitGraf has this syntax:

```
PROCEDURE InitGraf (globalPtr: QDPtr);
```

where globalPtr is thePort, a pointer to the first variable in QuickDraw's table of global variables. In C, the format is:

```
pascal void InitGraf(Ptr globalPtr);
```

In a Pascal program, this is always the format for calling InitGraf:

```
InitGraf(@thePort);
```

In C, you call InitGraf this way:

```
InitGraf((Ptr) &qd.thePort);
```

where globalPtr is thePort.

For an example of how InitGraf is used in an application, see the Creation program in Appendix C.

▶ Segmenting an Application

As you may recall from Chapter 6, every well-behaved application has a main event loop. In its main event loop, an application recognizes events such as keydown operations and mouse clicks, and it responds accordingly.

Since the main event loop is the most important part of a Macintosh application, it's essential that the program's main event loop remain in memory for as long as the application is running. However, most programs have certain parts—for example, sections that are used only for initialization purposes—that are used only once, or are used so rarely that they don't have to remain in memory all the time.

To make the best use of memory when you write an application, you can divide your program into segments: a main segment, which includes the main event loop and other portions of code that should remain in memory all the time, and other segments that are used just once or are used just now and then.

When you have divided a program into segments, you can launch your program in the usual way. You then use an operating system manager called the Segment Loader to unload segments of the program when they are no longer required. To free the memory occupied by a segment that's no longer needed (or that won't be needed for a while), all you have to do is call the Segment Loader routine UnloadSeg. There is also a LoadSeg call, and it's often used by the Macintosh system for segment management operations, but it's rarely called by applications.

The reason that programs don't often call LoadSeg is that they don't have to. When you free a block of memory using UnloadSeg, the segment you have designated doesn't go away forever; it's merely marked as purgeable. Therefore, if your application calls a routine in a segment that has been unloaded, the unloaded segment may still be in memory. If it isn't, the Segment Loader writes it back into memory automatically. So you never have to call LoadSeg explicitly, even to restore a segment that has been unloaded.

Furthermore, you can't break anything by calling UnloadSeg on a segment that has already been unloaded. The Segment Loader figures out that the segment has already been unloaded. Since UnloadSeg doesn't do anything when it's called on a segment that has already been unloaded, some applications go so far as to call UnloadSeg on rarely used segments with every iteration of the main event loop. That way, if a rarely needed segment has been loaded into memory, it will be unloaded the very next time the main event loop executes.

When you write a program using MPW Pascal, you can set up as many segments as you like. To figure out what routines belong in what segments, all you have to do is place routines that perform similar operations in the same segment. For example, procedures that perform disk operations could be collected together and unloaded (if necessary) with each execution of the main event loop.

In Pascal, this is the syntax of the UnloadSeg call:

```
PROCEDURE UnloadSeg (routineAddr: Ptr);
```

In C, this is the format:

```
pascal void UnloadSeg(Ptr RoutineAddr);
```

One excellent way to use UnloadSeg is to place all your program's initialization routines in the same segment and then unload that segment as soon as your program is initialized. For an example of a program that is set up this way, refer to the Creation.p program in Appendix C.

Once you have decided how you want to segment an application, you can easily place any procedure in the program in any segment you desire. In a program written in MPW Pascal, all you have to do is precede the name of a segment with the character combination "$S", and place it inside curly brackets in a line above a procedure's source code. For example,

```
{$S Main}
PROCEDURE AboutDialog;
```

When you write a program in MPW C, you can assign a function to a segment in a similar manner. Just type the words "#pragma segment," followed by the name of a segment, on a line that precedes the function's source code. For example,

```
#pragma segment Main
AboutDialog()
```

As you develop your program, you may find that you want to change the way it's segmented; you may find that you have assigned some procedures to segments where they don't really belong, and you may need to move them to segments where they fit better. That's no problem. If you want to move a routine from one segment to another, just change the segment designation that precedes the routine and recompile your application.

Warning ▶

Never, ever assign a routine to a segment, call a routine in another segment, and unload the calling routine in the routine that's called. If you do, the called routine won't have a valid memory location to return to!

One way to avoid making that kind of error is to call UnloadSeg only from your main program segment—which, if you've followed all the suggestions made so far, will never be unloaded from memory. In fact, it might be a good idea to make this an ironclad rule: *I will never call UnloadSeg from any segment except "main."*

Note ▶

The 32K Segment Boundary. In programs that were written for the earliest Macintosh computers—in which memory was severely limited—applications had to be broken up into segments that were no more than 32K long. It's no longer essential to limit segment lengths to 32K, but Macintosh development systems still work best with segments that are no longer than 32K, so the convention in writing Macintosh programs is still to set a 32K limit on segment length.

▶ Calling the Memory Manager

The four most often used Memory Manager calls are NewHandle, DisposHandle, NewPtr, and DisposPtr.

NewPtr, as noted earlier in this chapter, allocates a block in the heap of a requested size and returns a pointer to the block. You can then make as many copies of the pointer as you need and use them in any way your program requires. When you're finished with the block, you can free the space it occupies with the DisposPtr call.

NewHandle allocates a block in the heap of any size you have requested, and returns a handle to the block. You can then make as many copies of the handle as you need and use them in any way your program requires. When you're finished with the block, you can free the space it occupies with the DisposHandle call.

▶ Purging Memory Blocks

If the Memory Manager can't allocate a block of a requested size even after it compacts the heap, it can try to free some space by purging blocks from the heap. When a block is purged, it is removed from the heap, and the space it occupies is freed. The block's master pointer is set to nil, but the space occupied by the master pointer itself remains allocated. From then on, any handle associated with the block points to a nil master pointer and is said to be an empty handle.

If a program needs access to a relocatable block that has been purged, it can examine the status of the block's handle. If the handle contains a nil, that means that the handle has become empty, and the application can reuse it by making the ReallocHandle call. This call creates a new block using the same handle and updates its original master pointer, so that its handle is no longer nil and refers correctly to its new location.

When you obtain a memory allocation and a handle by using the NewHandle call, the block of memory associated with the handle is automatically designated nonpurgeable. You can change the status of a handle from unpurgeable to purgeable by making the HPurge call. Conversely, you can make a purgeable handle unpurgeable by making the HNoPurge call. Before you use HNoPurge, however, you should make sure that the block hasn't already been purged.

In practice, you'll rarely have to worry about any of this because the HPurge and HNoPurge calls are rarely used in well-behaved applications. There's usually a better way to use the Macintosh system's ability to purge blocks of relocatable memory. The method is this: When you want to make a block of memory purgeable, just make it a purgeable resource. That way, the Resource Manager will automatically purge and restore the block as necessary, and you'll never have to keep track of HPurge and HNoPurge calls.

With the exception of such structures as menus and cursors, which generally should stay in memory for as long as they are needed, most kinds of resources are designated as purgeable. Then they can be removed from memory and restored from memory, as required, by the Resource Manager. All you have to do to make a resource purgeable is to set the resPurgeable flag in its attributes field when you create it. This procedure was explained in Chapter 6.

▶ Other Properties of Memory Blocks

Once a handle has been allocated, and its status has been set to relocatable or nonrelocatable, that status can never be changed. However, a relocatable block can also be designated locked or unlocked, and purgeable or unpurgeable. Furthermore, an application can set and change these attributes, as necessary.

As we saw earlier, when you lock a block of memory, it can't be moved, even if the heap is compacted. You can later unlock the block, once more allowing the Memory Manager to move it during compaction. You can use the HLock and HUnlock calls to lock and unlock a block of memory.

A block can't be purged from memory unless it's relocatable, unlocked, and purgeable. A newly allocated relocatable block is initially both unlocked and unpurgeable.

Other Memory Manager calls

A number of other Memory Manager calls—some of which may come in handy from time to time—are listed in Table 7-2.

Table 7-2. Memory Manager calls

Call	*Function*
GetHandleSize	Returns the size of a memory block associated with a handle.
GetPtrSize	Returns the size of a memory block accessed with a pointer.
SetHandleSize	Changes the size of a memory block associated with a handle.
SetPtrSize	Changes the size of a memory block accessed with a pointer.
RecoverHandle	Returns a handle that points to a specified master pointer.
CompactMem	Compacts the current heap zone.
PurgeMem	Purges blocks from the current heap zone.
EmptyHandle	Purges a relocatable block from its heap zone and sets its master pointer to NIL, making it an empty handle.
ReallocHandle	Allocates a new relocatable block with a specified handle and a specified logical size, and updates the handle by setting its master pointer to point to the new block.
FreeMem	Returns the amount of free space in a heap zone.
MaxMem	Returns the size of the largest single free block and the maximum amount by which the zone can grow. MaxMem compacts the entire zone and purges all purgeable blocks.

Table 7-2. Memory Manager calls (continued)

Call	*Function*
SetGrowZone	Sets the current heap zone's grow zone function as designated by the growZone parameter. A NIL parameter value removes any grow zone function the zone may previously have had.
InitZone	Creates a new heap zone, initializes its header and trailer, and makes it the current zone.
GetZone	Returns a pointer to the curent heap zone.
SetZone	Sets the current heap zone to a specified zone.
SystemZone	Returns a pointer to the system heap zone.
ApplicZone	Returns a pointer to the original application heap zone.
HandleZone	Returns a pointer to the heap zone containing the relocatable block with a specified handle.
PtrZone	Returns a pointer to the heap zone containing the relocatable block with a specified pointer.
InitApplZone	Initializes an application's heap zone and makes it the current zone. The contents of any previous application zone are lost, all previously existing blocks in that zone are discarded, and the zone's grow zone function is set to NIL. InitApplZone is called by the Segment Loader when an application starts up; normally, you shouldn't need to call it.
SetApplBase	Changes the starting address of the application heap zone to a specified address, and then calls InitApplZone. SetApplBase is normally called only by the system itself; it's another procedure that you shouldn't need to call.
InitZone	Creates a new heap zone, initializes its header and trailer, and makes it the current zone.
GetApplLimit	Returns the current application heap limit. GetApplLimit can be used in conjunction with SetApplLimit, described below, to determine and then change the application heap limit.
SetApplLimit	Sets the application heap limit, beyond which the application heap can't be expanded. The actual expansion is not under the application program's control, but is done automatically by the Memory Manager when necessary to satisfy allocation requests. Only the original application zone can be expanded.

Table 7-2. Memory Manager calls (continued)

Call	*Function*
MaxApplZone	Expands the application heap zone to the application heap limit without purging any blocks currently in the zone. If the zone already extends to the limit, it is not changed.
MoreMasters	Allocates another block of master pointers in the current heap zone. This procedure is usually called very early in an application.

In addition to their normal results, many Memory Manager routines yield a result code that you can examine by calling the MemError function. Error codes returned by Memory Manager calls are provided in the descriptions of the calls in the "Memory Manager" chapters of *Inside Macintosh*.

▶ MultiFinder and the Memory Manager

Prior to the advent of MultiFinder, the Macintosh could execute only one application at a time. As a result, the memory architecture of the Macintosh was relatively simple, as shown in the memory map presented at the beginning of this chapter.

In the pre-MultiFinder era, RAM was divided into two main zones: a system zone and an application zone. The system zone, situated in the lowest area of memory, contained system global variables and a system heap.

The application heap, as you saw earlier in this chapter, is an area of memory that holds an application's code, along with all its resources such as menus, dialogs, and icons. In addition, objects that are created by a program—for example, blocks of text and various kinds of data structures—are kept in the application heap. The stack holds the application's local variables and a set of QuickDraw globals that are used when the application is running.

When you install System Software Version 7, or use MultiFinder while running System Software Versions 5 or 6, you can have multiple applications open at once. There is still a system zone, and it is still organized the same way it was prior to the introduction of MultiFinder. But each application has its own application zone, including an application heap and a stack, as shown in Figure 7-10. In Figure 7-10, three applications are open, sharing available memory.

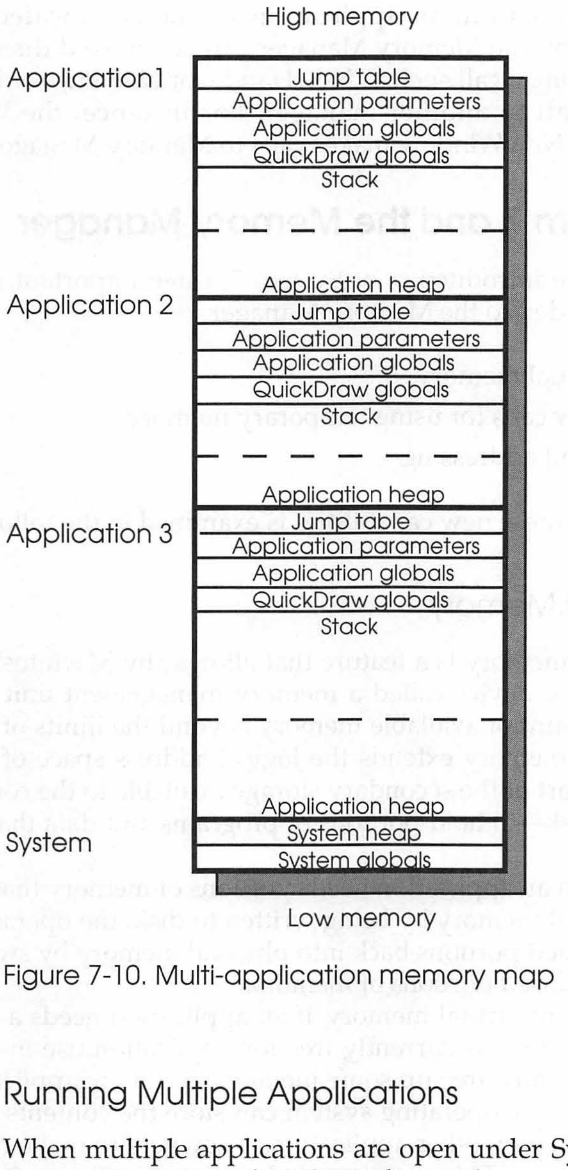


Figure 7-10. Multi-application memory map

▶ Running Multiple Applications

When multiple applications are open under System 7, or under the pre-System 7 version of MultiFinder, each open application allocates and frees space within its own application zone by making calls to the Memory Manager. Even when multiple applications are open, however, only one application can have control of the 680X0 processor at any given time. When a user selects an application as the foreground application, it becomes the active application and has control of the CPU.

Space within an application's zone is allocated by the Memory Manager. The Memory Manager can be invoked directly—for example, by making a call such as NewHandle or NewPtr—or indirectly, by making a call to another manager. For instance, the Window Manager routine NewWindow makes calls to Memory Manager routines.

System 7 and the Memory Manager

With the introduction of System 7, three important new features have been added to the Memory Manager.

- Virtual memory
- New calls for using temporary memory
- 32-bit addressing

Each of these new capabilities is examined in the following subsections.

Virtual Memory

Virtual memory is a feature that allows any Macintosh equipped with a hardware device called a memory management unit (MMU) to extend the amount of available memory beyond the limits of its physical RAM. Virtual memory extends the logical address space of the Macintosh by using part of the secondary storage available to the computer—such as a hard disk—to hold portions of programs and data that are not currently in use.

When an application needs portions of memory that have been placed in virtual memory by being written to disk, the operating system brings the needed portions back into physical memory by swapping them with other unused portions of memory.

Without virtual memory, if an application needs a greater amount of memory than is currently free for application use in the user's system, the user must free up some memory to run the application. With virtual memory, the operating system can store the contents of memory that is being used by other applications elsewhere to make room for the active application. This process of shuttling portions of memory between physical RAM and a secondary storage device is called paging.

Except for additional overhead caused by disk I/O, and the extra amount of storage required on secondary storage devices such as hard-disk drives, the operation of virtual memory is almost transparent to most Macintosh applications. Consequently, most applications do not have to know whether virtual memory is installed. However, the use of virtual memory can affect applications that have critical timing requirements, execute code at interrupt time, or perform debugging operations.

The use of virtual memory can result in a loss of processing speed, since it takes time to access paged-out segments of memory and pull them back into physical memory. This performance degradation ranges from unnoticeable to severe, depending on the ratio of virtual memory to physical RAM and the behavior of the actual applications running.

To the user, the use of virtual memory has two main benefits: More applications can be run at the same time, and applications can work with larger amounts of data than they are able to when logical address space is limited to available RAM. With virtual memory, instead of equipping a machine with amounts of RAM large enough to handle all possible needs, a user can install only enough RAM to meet average needs. Then, when more memory is occasionally needed for large tasks, virtual memory can provide the extra amount of memory required.

When virtual memory is present, the perceived amount of RAM can be extended to as much as 14 megabytes on existing systems and as much as 1 gigabyte on systems with 32-bit clean ROMs, such as the Macintosh IIci and the Macintosh IIfx.

Requirements for Using Virtual Memory

To use virtual memory, you must have two things: the right software and the right hardware. The software required to use virtual memory is System Software Version 7.0 or later. The hardware that is needed is a Macintosh equipped with a memory management unit, or MMU.

The Macintosh IIx, Macintosh IIcx, Macintosh IIci, Macintosh IIfx, and Macintosh SE/30 are ready to run virtual memory as soon as System Software Version 7 is installed. A Macintosh II can take advantage of virtual memory if it has a 68851 PMMU coprocessor on its main logic board in place of the standard address management unit (AMU). The 68851 PMMU, incidentally, is the same coprocessor that is needed to run A/UX.

Computers equipped with a Motorola 68000 processor, that is, the Macintosh Plus, the Macintosh SE, and the Macintosh Portable, cannot take advantage of the virtual memory capabilities of System 7. However, they can run System 7—and take advantage of many of its other capabilities—provided they have at least 2 MB of RAM. Furthermore, owners of Macintosh SEs have the option of upgrading their machines to Macintosh SE/30s.

How Virtual Memory Is Allocated

The Macintosh user can control and configure virtual memory by using System 7's memory control panel. This panel provides controls that allow the user to turn virtual memory on or off, set the size of virtual memory, and set the disk volume on which the backing store resides. On the disk volume that is selected, the user can also choose the file that the Operating System uses to store the contents of nonresident portions of memory.

The memory control panel also provides several other memory-related user controls. For example, the user can set up a disk cache and can select 24-bit or 32-bit Memory Manager addressing if the Macintosh being used is a model that permits it.

System 7 includes a number of new Memory Manager calls that are used in connection with virtual memory. These calls are listed and described in *Inside Macintosh,* Volume VI.

▶ Temporary Memory

When you write an application designed to be run under System Software Version 6 or 7, you can use a feature called temporary memory to allocate extra memory to the application for limited periods of time. With the introduction of System 7, seven new routines for managing temporary memory allocation were added to the Memory Manager. By using these routines, an application can request additional memory for occasional short-term needs.

When you write a program that takes advantage of System 7's temporary memory feature, any available memory—that is, any memory that is not currently allocated to another application's RAM partition—is available for use by your application upon demand. When your application no longer needs the temporary memory it has been using, it can release the memory, making it available for use by other applications or by the operating system.

When you have System 7 installed, the Macintosh operating system also makes use of temporary memory management routines. For example, the Finder uses System 7's temporary memory feature to secure buffer space to be used during file copy operations.

To determine how temporary memory is allocated to an application, the operating system uses the application's 'SIZE' resource. As explained in Chapter 6, every MultiFinder-aware application has a 'SIZE' resource. Among other things, an application's 'SIZE' resource specifies how much memory it requires, and what kinds of hardware and system software it is designed to run under.

When you launch an application, the amount of memory allocated to it is set to the preferred size specified in its 'SIZE' resource if that much contiguous memory is available. Otherwise, it is set to some smaller size. However, the amount of memory allocated to an application is never smaller than the minimum size specified in the application's 'SIZE' resource.

Prior to the introduction of System 7, it was customary for an application to use the 'SIZE' resource to request the largest amount of memory that it might ever need for its application heap. This large amount of memory was specified as the preferred partition size in the application's 'SIZE' resource.

With the advent of System 7, it is no longer necessary to specify such a large preferred memory allocation. When you create an application designed to be run under System 7, you should specify a smaller (but reasonable) preferred partition size. When you need more memory than that for temporary use, you can use the temporary memory allocation capabilities provided by the operating system under System 7.

Because the amount of temporary memory you request might not always be available, however, you should not rely on always getting the memory you need every time you issue a temporary memory request. So you should still make sure that your application will work even if there is no temporary memory available when you request it.

One example of application designed in this way is the System 7 Finder. If the System 7 Finder needs to copy a file but it can't allocate a large temporary copy buffer, it performs the copy using a small reserved copy buffer situated within its own heap zone. Thus, although the copy might take longer than it would using temporary memory, it is still performed.

Temporary Memory in System 6 and System 7

The seven new temporary memory routines introduced in System 7 are as follows.

- TempFreeMem
- TempMaxMem
- TempDisposHandle
- TempHLock
- TempHUnlock
- TempNewHandle
- TempTopMem

System Software Version 6 had seven similar routines: MFFreeMem, MFMaxMem, MFTempDisposHandle, MFTempHLock, MFTempHUn lock, MFTempNewHandle, and MFTopMem. For compatibility, you can continue to use these names when you write programs using System 7.

Under System Software Version 7, the following Memory Manager routines work even if the handle or pointer was allocated by a temporary memory routine.

- DisposHandle
- EmptyHandle
- GetHandleSize
- HandleZone
- HClrRBit
- HGetState
- HLock
- HNoPurge
- HPurge
- HSetRBit
- HSetState
- HUnlock
- ReallocHandle
- RecoverHandle
- SetHandleSize

Note that the pre–System 7 calls TempDisposHandle, TempHLock, and TempHUnlock are obsolete under System 7, although they still work (for the sake of compatibility).

For more details on using the temporary memory calls introduced with System 7, see *Inside Macintosh,* Volume VI.

▶ Conclusion

This chapter explained how the Memory Manager is used in programs written using MPW, and presented some new features offered by the Memory Manager in System Software Version 7.

If you have carefully studied this chapter and the two chapters that preceded it, you now know how to use the "Big Three" Macintosh managers: the Event Manager, the Resource Manager, and the Memory Manager.

8 ▶ Building an Application

Here, at last, is the chapter in which you'll finally get a chance to write, build, and execute an application program. In Part One, you learned how to write scripts and source code using the MPW editor. Then, in Chapters 5, 6, and 7, you learned how to use the "Big Three" Macintosh managers: the Event Manager, the Resource Manager, and the Memory Manager. In this chapter, you'll have an opportunity to put all this knowledge together in a very practical way: by compiling, linking, and executing an application program.

▶ Building a Program with MPW

In MPW, the process of compiling and linking a program is known as building the program. To build an application in the MPW environment, you must take these steps:

1. Create the resource code for the program using the MPW editor.
2. Write a resource fork for the program using the MPW editor and Rez, or ResEdit. If you write your resource fork using ResEdit, you can decompile it into source code using MPW's DeRez command.
3. Compile or assemble your program. If your program is written in MPW Pascal, MPW C, or MPW assembly language, you can compile it using the Pascal command, the C command, or the Asm command. Other commands may be used to compile programs written in other MPW-compatible languages.

4. If you have written your program's resource fork using the MPW editor, you must compile it using the MPW command DeRez.
5. You can then link your program using the Link command. Link resolves cross-references between the segments in a program and links object files—including object-code segments, library files, and resource forks—into an executable program. One powerful feature of the MPW linker is that it can link programs with segments that are written in various MPW languages.

Procedures for compiling and decompiling resources were described in Chapter 6. The other steps in the preceding list are covered in this chapter.

Note ▶

Is It "Assemble" or "Compile"? Programmers who work with MPW often speak of "compiling" assembly language programs, rather than "assembling" them. That isn't precisely correct—or is it?

As every assembly language programmer knows, assemblers assemble programs and compilers compile them. However, the MPW assembler is a little different from most other assemblers; it produces object-code segments that are constructed exactly like the object segments created by MPW's Pascal and C compilers. So, once you've created an object-code segment using the MPW assembler, the MPW linker will happily link with a C or Pascal segment, without knowing or caring that it was produced by an assembler rather than a compiler.

Since the MPW assembler works so much like a compiler, the distinction between assembling a program with an assembler and compiling it with a compiler has become somewhat blurred in the MPW world. So, if you get a little sloppy and start talking about "compiling" a program that's written in MPW assembly language, MPW wizards will probably never notice. They make the same mistake (if it is one) all the time.

▶ Three Ways to Build a Program

Although it's possible to build a program by typing command lines that compile and link it, there is an easier way. In fact, there are two other ways.

One way is to simplify the build process by using the MPW commands CreateMake and Make. There are three steps in using the CreateMake and Make commands:

- The CreateMake command creates a makefile, a script that contains rules for building a program.
- You expand these rules into a set of commands that build a program by executing the Make command.
- You build the program by executing the build commands generated by the Make command.

An even easier way to build a program is to select the items "Create Build Commands" and "Build" from the MPW Build menu.

The three methods that you can use to build a program—the command-line method, the write-your-own-makefile method, and the quick and easy menu method—are illustrated in Figure 8-1.

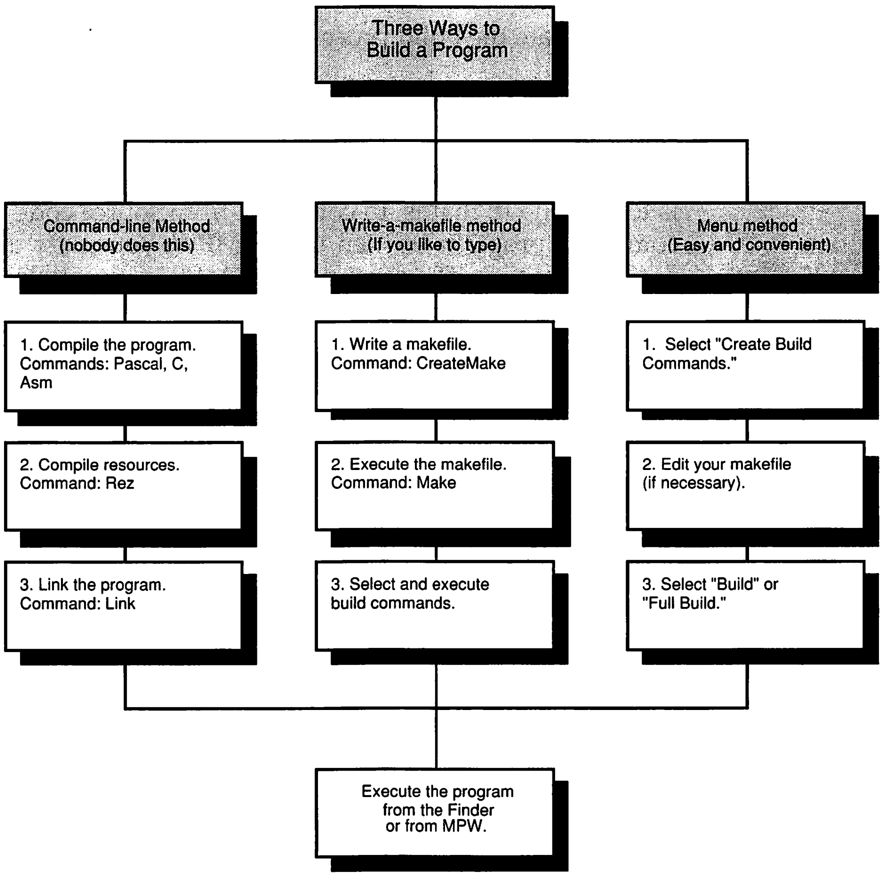


Figure 8-1. Three ways to build a program

If you decide to build a program using the write-your-own-makefile method, you must write and execute a CreateMake command and a Make command. When you build a program using the MPW Build menu, MPW takes care of executing the necessary Make and CreateMake commands.

▶ What You'll Learn in this Chapter

In this chapter, you will learn how to

- compile (or assemble) an application using the Pascal, C, and Asm commands
- link an application using the Link command
- create an object-code library using the Lib command
- simplify the build process by using the CreateMake and Make commands
- automate the build process by using the MPW Build menu

▶ Compiling an Application

The first step in building a program written in MPW Pascal or MPW C is to compile it; that is, to convert it from source code to object code. To compile a program written under MPW, you must use an MPW-compatible compiler. The most popular MPW-compatible compilers are the MPW Pascal compiler and the MPW C compiler.

▶ The MPW C and Pascal Compilers

The MPW Pasacal compiler and the MPW C compiler are manufactured by Apple and are designed to be used with MPW. They are not provided with the basic MPW system, but must be purchased separately. However, packages that include the MPW system and the compiler of your choice—or that include both compilers—are available as specially priced "bundles" from APDA, the Apple Programmer's and Developer's Association. (APDA's address is provided in the Preface.)

▶ The MPW Assembler

Apple also manufactures an assembler that can be used to assemble assembly language programs written under MPW. The MPW assembler is also offered by APDA as a separate product; but it, too, can be purchased bundled with the MPW system. In fact, the MPW system, the MPW assembler, and both MPW compilers are available all packaged together in one giant bundle.

▶ The MPW C Compiler

The MPW C compiler package includes

- the MPW C compiler
- the standard C library
- C interfaces to Toolbox and operating system functions
- sample programs written in MPW C

A C++ translator, which can be used with the C compiler package, is available from APDA.

To use the MPW C compiler, you must install it on your hard disk using the procedures described in the compiler's documentation and outlined in Chapter 2. You can then write a C program using the MPW editor and compile it using the MPW command C.

You can execute the C command from a command line or a Commando dialog; however, a much more common (and convenient) way to use it is to include it in a makefile, as explained in this chapter.

▶ The C Command

When you write a source file in C, MPW expects it to have a name with the suffix ".c". Hence, if you wanted to write a C program called Creation, MPW would expect you to give the program's source file the name Creation.c.

You can compile a program written in C by using the C command. The syntax of the C command is:

```
C [option...] [file.c]
```

When you execute the C command, the MPW C compiler compiles the source file specified in the file parameter, and creates an object file named file.c.o. Thus, the MPW command

```
C Creation.c
```

compiles the C language source file Creation.c into an object file named Creation.c.o.

If you don't specify a file parameter when you execute the C command, MPW reads standard input—usually the screen—and compiles an object file named c.o.

(As the MPW C compiler processes a program, it also creates a temporary, or intermediate, file named file.c.i. This file is used internally by the compiler; you don't have to be concerned with its contents when you build a program.)

When the compiler encounters an error, it writes an error message to diagnostic output, usually the screen.

The -p and -e Options

If you use the -p option, as in the command

```
C -p Creation.c
```

the compiler writes progress information (including file names, function names, and sizes) and summary information (including the number of errors and warnings, code size, global data size, and compilation time) to diagnostic output. This option generates and displays very useful information and is often used with the C command.

If you use the -e option or the -e2 option, preprocessor output is written to standard output, but no object file is created.

Options Used with the C Command

These options can be used with the C command.

Option	*Function*
-b	Puts string constants into code and generates program counter-relative references.
-b2	Same as -b, but also allows the code generator to reduce code size by overlaying string constants whenever possible.
-b3	Allows the code generator to keep string constants in the code segment and overlays them when possible, but always generates program counter-relative references for function addresses.

Option	*Function*
-c	Syntax check only; doesn't create object file.
-d *name*	Equivalent to "#define *name* 1".
-d name=*string*	Equivalent to "#define name *string*".
-e	Writes preprocessor results to output.
-e2	Same as -e, but strips comments.
-elems881	Generates MC68881 code for transcendentals.
-i *directory* [, *directory*]...	Searches for includes in specified directory or directories.
-m	Generates 32-bit references for data (produces less efficient code).
-mbg ch8	Includes MPW 2.0–compatible MacsBug symbols in code.
-mbg *n*	Includes MacsBug symbols truncated to length *n* (*n* can be any number from 0 through 255).
-mbg off	Doesn't include MacsBug symbols in the code.
-mbg on | full	Includes full (untruncated) MacsBug symbols in code.
-mc68020	Generates code that takes advantage of the 68020 processor.
-mc68881	Generates code that takes advantage of the 68881 coprocessor for arithmetic operations.
-n	Changes errors associated with pointer assignment incompatibilty into warnings.
-o *objname*	Generates code in file or directory *objname*.
-p	Writes progress information (including file names, function names, and sizes) and summary information (the number of errors and warnings, code size, global data size, and compilation time) to diagnostic output. This option generates and displays useful information and is often used with the C command.
-r	Outputs a warning when attempting to call a function that has no definition.
-s *n*	Names the main code segment *n* (default name is main).

Option	*Function*
-sym off	Does not generate SADE object file information.
-sym on | full	Generates full SADE object file information; this option can be modified with [,nolines], [,notypes], and [,novars].
-t	Writes compilation time to diagnostic output.
-u *name*	Equivalent to "#undef *name*".
-w	Suppresses warnings.
-w2	Outputs additional warnings (about constructs that the compiler has reason to suspect).
-y *directory*	Puts the compiler's intermediate ("o.i") files into the specified *directory*.

Status codes returned by the C command are as follows.

Status Code	*Meaning*
0	Successful completion.
1	Errors occurred.

▶ The MPW Pascal Compiler

The MPW Object Pascal system includes

- a compiler that produces object code for programs written in Pascal and Object Pascal
- PasMat, a utility for converting Pascal source code into a standard format suitable for generating printouts or compilation listings
- PasRef, a tool for cross-referencing identifiers and printing lists of cross-references
- ProcNames, a tool for listing the names of procedures and functions used in Pascal programs
- a Pascal runtime library
- Pascal interfaces for Toolbox and operating system routines and functions
- sample programs written in Pascal

The MPW Pascal compiler is compatible with MacApp, Apple's powerful class library for writing object-oriented application programs.

Before you can use the MPW Pascal compiler, you must install it on your hard disk using the procedures described in the compiler's documentation. You can then write a Pascal program using the MPW Editor and compile it using the MPW command Pascal.

The Pascal command can be executed from a command line or a Commando dialog; however, it is usually issued from a makefile, as explained later in this chapter.

▶ The Pascal Command

The syntax of the Pascal command is:

```
Pascal [option...] [file.]
```

Source files written in MPW Pascal have the suffix ".p". Thus, if you wanted to write a Pascal program called Creation, MPW would expect you to give the program's source file the name Creation.p.

When you execute the Pascal command, the MPW Pascal compiler compiles the source file (either a program or a unit) specified in the file parameter and creates an object file named file.p.o. Thus the MPW command

```
Pascal Creation.p
```

compiles the Pascal source file Creation.p into an object file named Creation.p.o.

If you don't specify a file parameter when you execute the Pascal command, MPW reads standard input—usually the screen—and compiles an object file named p.o.

(As the MPW Pascal compiler processes a program, it also creates a temporary, or intermediate, file named file.p.i. This file is used internally by the compiler; you don't have to be concerned with its contents when you build a program.)

When the compiler encounters an error, it writes an error message to diagnostic output, usually the screen.

The -p and -e Options

If you use the -p option, as in the command

```
Pascal -p Creation.p
```

the Pascal compiler writes progress information and summary information (as noted in the following options listing) to diagnostic output.

Options Used with the Pascal Command

These options can be used with the Pascal command.

Option	*Function*
-b	Generates A5-relative references whenever the address of a procedure or function is taken. (By default, program counter-relative references are generated for routines in the same segment.)
-c	Syntax check only; doesn't create object file.
-clean	Erases all symbol table references.
-d *name*	Equivalent to "#define *name* 1".
-d name=TRUE | FALSE	Sets the compile time variable *name* to TRUE or FALSE.
-e *errLogFile*	Writes all errors to the error log file *errLogFile*. A copy of the error report is still sent to diagnostic output.
-forward	Allows only forward and external object declarations.
-h	Suppresses error messages regarding the use of unsafe handles.
-i *directory* [, *directory*]...	Searches for include or USES files in the specified directories. Multiple -i options may be specified. At most, 15 directories are searched. (For the order in which directories are searched, see the Pascal command in the *MPW 3.0 Reference*, Volume II.)
-m	Allows globals larger than 32K.

Option	*Function*
-mbg ch8	Includes MPW 2.0-compatible MacsBug symbols in code.
-mbg full	Includes full (untruncated) MacsBug symbols in code.
-mbg *n*	Includes MacsBug symbols truncated to length *n* (*n* can be any number from 0 through 255).
-mbg off	Doesn't include MacsBug symbols in the code.
-mc68020	Generates code that takes advantage of the 68020 processor.
-mc68881	Generates code that takes advantage of the 68881 coprocessor for arithmetic operations.
-n	Generates separate global data modules for better allocation.
-noload	Doesn't use or create any symbol table resources.
-o *objname*	Generates code in file or directory *objname*.
-only *name*	Generates code only for named modules.
-ov	Turns on overflow checking.
-p	Writes progress information (including module names, code sizes in bytes, number of errors, and compilation time) and summary information (including compiler header information, that is, copyright notice and version number) to diagnostic output.
-r	Suppresses range checking.
-rebuild	Rebuilds all symbol table references.
-roger	Reallocates unused scratch registers for local data.
-sl	Omits static links for nested procs that do not need them.
-sym off	Does not generate SADE object file information.
-sym [on | full]	Generates full SADE object file information; this option can be modified with [,nolines], [,notypes], and [,novars].

Option	*Function*
-t	Writes compilation time to diagnostic output. (The -p option also reports compilation time.)
-u	Initializes local and global data to the value $7267 (hexadecimal 7267) for use in debugging.
-w	Turns off the compiler's "peephole optimizer."
-y *directory*	Puts the compiler's intermediate files into the specified directory.

Status codes returned by the Pascal command are:

Status Code	*Meaning*
0	Successful completion.
1	Error in parameters.
2	Error halted compilation.

▶ The MPW Assembler

The MPW assembler package includes

- an assembler that can translate programs written for the MC68000, MC68020, and MC68030 processors into object code
- support for the MC68881 and MC68882 math coprocessors, as well as the MC68885 memory management unit
- macro facilities, code and data modules, support for entry points, local labels, and (optional) optimized instruction selection
- assembly language interface to Toolbox and operating system routines
- sample programs written in MPW assembly language

To install the MPW assembler on your hard disk, use the procedures described in the assembler's documentation and outlined in Chapter 2. When you have installed the assembler, you can write an assembly language program using the MPW editor and then assemble it using the MPW command Asm.

To assemble a program written using the MPW assembler, you must use the Asm command. You can issue the Asm command from a command line or a Commando dialog; however, the command is usually executed from a makefile, as explained later in this chapter.

▶ The Asm Command

Source files written in MPW assembly language have names that end with the suffix ".a". So, if you wanted to write an assembly language program called Creation, MPW would expect you to give the program's source file the name Creation.a.

The syntax of the Asm command is:

```
Asm [option...] [file.a]
```

When the Asm command is executed, the MPW assembler assembles the source file specified in the file parameter and creates an object file named file.a.o. Thus the command

```
Asm Creation.a
```

assembles the assembly language source file Creation.a into an object file named Creation.a.o.

If you don't specify a file parameter when you execute the Asm command, MPW reads standard input—usually the screen—and assembles an object file named a.o.

When the assembler encounters an error, it writes an error message to diagnostic output, usually the screen.

▶ The -o Option

If you use the -o option in this format:

```
Asm -o [differentName] sourceName.a
```

the assembler gives the object file that it creates the name differentName, instead of the name sourceName.

▶ Options Used with the Asm Command

Option	*Function*
-addrsize n	Sets the size of the address display to *n* digits (4 through 8 digits allowed). The default is 5.
-blksize blocks	Sets the assembler's text file I/O buffers size to blocks times 512 bytes. The default is 16 (8192 bytes) if the assembler can find enough memory space; otherwise, the default is 6 (3072 bytes).
-case obj[ect]	Preserves the case of module, EXPORT, IMPORT, and ENTRY names only in the generated object file. In all other respects, behavior is the same as in the default -case off setting.
-case off	Ignores the case of letters (default setting).
-case on	Same as the CASE ON directive; causes the assembler to distinguish between uppercase and lowercase letters in nonmacro names. The default is -case off.
-c[heck]	Syntax check only; doesn't create object file.
-d[efine] *name*	Defines the macro *name* having the value 1 (same as the name EQU 1 directive).
-d[efine] *name=v*	Defines the macro *name* as having the value *v* (same as the name EQU *v* directive).
-d[efine] &name	Same as the &name SET[AC] 1 directive.
-d[efine] &name=*value*	Same as the directive &name SET[AC] *value*.
-e[rrlog] *fileName*	Writes errors and warnings to *fileName* (same as the ERRLOG 'fileName' directive).
-f	Suppresses page ejects in listing (same as the PRINT NOPAGE directive).
-font *fontName*[, *typeSize*]	Prints listing in *fontName* and *typeSize.*
-h	Suppresses page headers in listing (same as the PRINT NOHDR directive).
-i *directory*[, *directory*]...	Searches for include and load files in specified directory or directories. A maximum of 15 directories can be searched.
-l	Writes full listing to output.
-lo *pathName*	Writes listing of output to specified file or directory.

Option	*Function*
-o *objName*	Generates code in file or directory *objName*.
-p	Writes assembly progress information (module names, includes, loads, and dumps) and summary information (number of errors, warnings, and compilation time) to the diagnostic output file. Same as the PRINT STAT directive.
-pagesize [ll[, w]	Sets page length and width, in dots, for printed listing. The default length is 75; the default width is 126. The defaults assume that listing is printed in 7-point Monaco type.
-print *mode*	Same as the PRINT *mode* directive. For lists and descriptions of options, see the *MPW 3.0 Reference* and the *MPW 3.0 Assembler Reference.*
-s	Generates a shortened form of the printed listing.
-sym off	Does not write object file records containing information for the SADE debugger.
-sym on | full	Writes object file records containing information for the SADE debugger. You can modify this option with [,nolines], [,notypes], and [,novars].
-t	Writes assembly time and the number of lines generated to diagnostic output.
-w	Suppresses warning messages (same as the PRINT NOWARN directive).
-wb	Suppresses warning messages related to branch instructions.

Status codes returned by the ASM command are as follows.

Status Code	*Meaning*
0	No errors detected in any of the files assembled.
1	Parameter or option errors.
2	Errors detected.

Using Multiple Options with the Asm Command

You can use multiple options with the Asm command. For example, the command

```
Asm -w -l Creation.a Menu.a -d Debug
```

assembles the source files Creation.a and Menu.a, suppresses warnings, defines the name Debug as having the value 1, and generates two listing files: Creation.a.lst and Menu.a.lst. Two object files are produced: Sample.a.o and Memory.a.o.

Linking an Application

When you have compiled an MPW C, Pascal, or assembly language program—or a program containing segments written in two or more MPW-compatible languages—the next step in building the program is to link it using the MPW linker.

The MPW Linker

The MPW linker is a tool that is included in the basic MPW package. You can invoke the linker by executing the MPW command Link. The Link command can be issued from a command line, a Commando dialog, or a makefile. In most circumstances, the Link command is produced by a makefile. More information about makefiles is presented later in this chapter.

The MPW linker does its work by combining various kinds of raw object-code modules and segments into various kinds of linked code.

A module is a unit that contains code or static data. It is the smallest unit that can be manipulated by the linker. Raw object-code modules can include object-code libraries and compiled or assembled object-code files. A segment is a named collection of modules.

The raw object code that the linker accepts as input can include object-code libraries and compiled or assembled source-code modules and segments. Raw object code produced by a compiler or an assembler contains object code that has been compiled or assembled into relocatable machine language, and symbolic references to identifiers whose locations were not known at compile time.

Libraries that can be linked with compiled or assembled segments and modules are listed in Table 8-1. It is a good practice to link newly written programs with all the libraries that they may need. If it turns out that the libraries are not needed, the linker omits them from linked code and can produce listings of the libraries that are not needed, as explained later in this chapter.

Table 8-1. Libraries that can be used by the MPW linker

Library	*Type of Library*	*Comments*
{Libraries}Interface.o	Toolbox interface	Contains interfaces for the Toolbox and the operating system.
{Libraries}Runtime.o	Non-C library	Provides runtime support; use if no part of your program is written in C. Do not use if you are using {CLibraries}
{PLibraries}PasLib.o	Pascal library	Use with programs written in Pascal.
{PLibraries}SANELib.o	Pascal library	SANE numerics library.
{CLibraries}CSANELib.o	C library	SANE numerics library.
{CLibraries}Math.o	C library	Math functions.
{CLibraries}StdCLib.o	C library	Standard C library.
{Libraries}ObjLib.o	Specialized library	Object-oriented programming library (Pascal and assembler).
{Libraries}ToolLib.o	Specialized library	Routines for MPW tools.
{Libraries}DRVRRuntime.	For driver resources	Driver runtime library.

▶ The Linker and Resources

The linker produces linked code in the form of resources. For example, the linker can generate an application program (resource type 'CODE'), an MPW tool (resource type 'CODE'), file type MPST, a driver resource (resource type 'DRVR'), or a standalone code resource such as a window definition procedure ('WDEF') or a control definition procedure ('CDEF'). The kind of resource that the linker produces depends on the options used with the Link command.

Note ▶

About Standalone Code Resources. A standalone code resource is a resource that is built separately from an application, and thus can be used with any application. Some examples of standalone code resources are:

Resource	*Description*
WDEF	Window definition procedure (for custom windows)
CDEF	Control definition procedure (for custom controls)
LDEF	List definition procedure (for the List Manager)
MDEF	Menu definition procedure (for custom menus)
INIT	Init resource (a resource that is loaded and run at boot time by the system startup code)
XFCN	An external function written for HyperCard
XCMD	An external command written for HyperCard

You must follow certain rules and guidelines when you write standalone code resources. For more information, see *Inside Macintosh*, the HyperCard references in the Apple Technical Library, and the *MPW 3.0 Reference.*

When the linker creates an executable file; it assigns the file a type and a creator. File types and creators are listed in Table 8-2.

Table 8-2. File types and creators

Kind of program	*Type*	*Creator*
Application	'APPL'	Developer-defined
MPW tool	'MPST'	'MPS'
Device driver	Varies	Developer-defined
Desk accessory	'DFIL'	DMOV
Script	'TEXT'	Developer-defined
Standalone resources	Varies	Developer-defined

Important ▶

Desk Accessories Are Dead; Long Live Desk Accessories. This book provides no instructions for writing and building desk accessories because, under System Software Version 7, desk accessories as we knew them have begun a slow but inevitable slide into oblivion. Under System 7, old-fashioned desk accessories still work, but you can also make any application work like a desk accessory. All you have to do is move the program's icon into the "Apple Menu Items" folder that resides in the System Folder. Thus it is no longer necessary to go through the hassle of writing one of those second-class applications (with limited size and with no global variables) that you once had to create if you wanted to write a desk accessory.

More information about the kinds of resources generated by the linker is presented later in this chapter.

When you use the MPW linker to link an application program, it links the program's object-code segments with any needed library routines and places them in 'CODE' segments in the program's resource fork. All existing 'CODE' segments are replaced, without disturbing any other resources in the program's resource file. The linker also resolves symbolic references and controls final program segmentation.

Note ▶

New Link and Lib Tools in MPW 3.2. A new linker and a new Lib tool were introduced with the unveiling of MPW 3.2. The new Link and Lib tools run faster than the old ones did, and they also have some new features, including:

- Better performance when the -sym on option is used (options are listed later in this chapter). In some cases, especially when large links are processed, the new tools yield much better performance. There is not as much difference in performance with smaller links, especially with links of less than 500K.
- Better compression of data initialization information used with C++ variables.
- A number of minor bug fixes.

The Link Command

You can link a program by executing the Link command. The syntax of the Link command is:

```
Link [option...] objectFile...
```

where objectFile is the name of the output (linked) file. The input object files must have type 'OBJ '. The output file is named Link.Out, by default, but you can specify a different file name by using the -o option.

When the linker links an application, it places the code segments that it generates in 'CODE' resources. All old 'CODE' resources are deleted before new 'CODE' resources are written.

How the Linker Works

During the course of a link, the linker performs these functions:

- Sorts code and data modules into segments, arranged by segment name. Within a segment, modules are placed in the order in which they occur in the input files. You can change the order of segments at link time by using the -sg and -sn options, as explained in the list of options presented later in this chapter.
- Automatically omits unused, or "dead," code and data modules from the output file. You can instruct the linker to list omitted modules by using the Link command's -uf option, and you can delete dead modules from libraries by using the -df option.
- Creates a jump table that the Segment Loader uses to relocate segmented code and data at runtime.
- Constructs jump table entries only when needed; that is, only when a symbol is referenced across segments. This ensures that the jump table created by the linker is no larger than is necessary.
- Edits instructions when necessary to use the most efficient addressing mode.
- When the -x option is specified, generates a listing of cross-referenced names at link time.
- When the -map option is specified, generates a location map for debugging or for performance analysis.

- Provides support for relocation of data references at runtime. Data references are relocated with the help of a module called the data initialization interpreter. This module, named _DATAINIT, is included in the Runtime.o and CRuntime.o libraries.

▶ Options Used with the Link Command

These options can be used with the Link command.

Option	*Function*
-ac *n*	Aligns code modules to *n* byte boundaries. The *n* argument must be a power of 2. The default is 2.
-ad *n*	Aligns data modules to *n* byte boundaries. The *n* argument must be a power of 2. The default is 2.
-c *creator*	Sets creator of file to *creator*. Default creator is '????'.
-d	Suppresses warnings about duplicate symbol definitions (for data and code).
-da	Converts segment names to desk accessory names at output time. Desk accessory names begin with a leading null character ($00). Use this option when you want to create a desk accessory (resource type 'DRVR').
-f	Treats duplicate data definitions as FORTRAN "common" regions; that is, multiple data modules with the same name. The size of the largest module is used. There may be no more than one initialization of the data.
-l	Writes a location-ordered map to standard output. MPW's performance-measurement tools and other scripts may rely on this option. Usually, the option is used with output redirection in effect. For example, the command Link TheObjFile -l > TheMapFile writes a location-ordered map to the TheMapFile file.
-la	Lists anonymous symbols in the location map. The default is not to list anonymous symbols.
-lf	Writes a location map to standard output and includes the symbol definition location in the input file, that is, the file number and byte offset of the module or entry-point record. The default is to omit the symbol definition locations.

Option	*Function*
-m *mainEntry*	Uses *mainEntry* as main entry point.
-ma *n=alias*	Gives the module or entry point *n* the alternate name *alias*. The option lets you resolve undefined external symbols at link time, when the problem is caused by differences in spelling or capitalization. Note that you cannot use an alias specification to override an existing module or entry point because the original name is retained.
-map	Writes a location map to standard output, but prints a more readable map so that the A5 world has the correct offsets. This option also provides sizes of all modules.
-mf	During the linking process (not when the linked program executes), uses MultiFinder's temporary memory allocation routines if they are available. If MultiFinder is not available, this option has no effect. If Link is in danger of running out of space in the MPW shell's heap, and if the extra memory is available, Link spills over into MultiFinder's temporary allocation region. This can cause a system crash if Link aborts abnormally, so use this option with caution.
-msg *keyword* [, *keyword*]	Enables or suppresses certain warning messages. This option can be used with three parameters. The parameter [no]dup enables or suppresses warnings about duplicate symbols; the parameter [no]multiple enables or suppresses multiple undefined symbol reports; and the parameter [no]warn enables or suppresses warning messages.
-o *outputFile*	Writes output to the file *outputFile*. If no outputFile is specified, Link's output file is named Link.Out.
-opt [*keyword...*]	Optimizes Object Pascal optimizations. This option is followed by one or more keywords. The default setting is option off. Other keywords are [on], which enables Object Pascal optimizations; [NoBypass], which enables optimizations but does not optimize monomorphic method calls to program counter-relative JMP instructions; [,Names], which embeds SelectorProc names; and [,MBgNames], which embeds MacsBug-visible SelectorProc names.

Option	*Function*
-p	Writes progress and summary information to diagnostic output.
-ra [seg]=*attr* [,*attr*...]	Sets resource attributes of the segment or segments being linked. If seg is specified, the single segment named seg is given the attribute value attr. (To set the attributes of all segments, you must specify this option before using any other options that name segments, such as -sn and -sg.) The segment containing the main entry point (the 'CODE' resource, which has an ID of 1) must be set individually to override default resource attributes. The *attr* parameter of this option can be expressed as a decimal or a hexadecimal number, or as a constant. Constants that can be used are resSysHeap (or simply sysHeap) resPurgeable (or Purgeable), resLocked (or locked), resProtected (or protected), resPreload (or preload), and resChanged (or changed). Although resChanged is a legal attribute, it has no effect. Resource attributes—along with the numeric values that can be used in place of their names—are described in Chapter 6.
-rn	Suppresses the name of resources (by default, each resource in a file has the name of the segment in which it is situated). Desk accessories must always be named.
-rt type=*ID*	Sets the output resource type to type and the resource *ID* to *ID*. The default setting for the type parameter of this option is 'CODE' (an application program); resource IDs are numbered from 0.
-sg *newSeg*= *old*[,*old*]...	Merges all code in the *old* segment or segments specified into a new segment named *newSeg*. If you do not specify any old segments, Link maps all segments to *newSeg*.
-sn *oldSeg*=*newSeg*	Changes segment name *oldSeg* to *newSeg*.
-srt	Sorts A5-relative data into 32-bit and 16-bit words. All 16-bit referenced data is placed as close as possible to the address pointed to by the A5 register (For more information about the A5 register, see Chapter 7.)

Option	*Function*
-ss size	Changes the maximum segment size to size. The default size is 32,760 bytes (32K minus a few overhead bytes). The size parameter can be set to any value greater than 32,760. It is not recommended that you use this option under normal circumstances since code segments larger than 32K will not load correctly on Macintosh models with 64K ROM, and they may not compile and link as efficiently as segments smaller than 32K.
-sym [off | on | full] [, *keyword*...]	Enables or disables the writing of symbolic data to support the SADE debugger. The default is -sym off. Keywords that can be used with the keyword option are [,NoLabels], which omits label information; [,NoLines], which omits source line information; [,NoTypes] which omits type information; and [,NoVars], which omits variable information.
-t *type*	Sets the type of the output file to *type*. The default type is 'APPL' (application program).
-uf *unrefFile*	Lists unreferenced modules in the *unrefFile* file. This option can be used to identify unused, or dead, source code.
-w	Suppresses warning messages.
-x *crossRefFile*	Writes cross-references to the *crossRefFile* file.

Status codes returned by the Link command are as follows.

Status Code	*Meaning*
0	No error detected.
1	Syntax error.
2	Fatal error.

▶ Using Multiple Options with the Link Command

The Link command has so many useful options that Link statements can be quite complex. This is a relatively simple Link statement:

```
Link Creation.p.o ∂
"{PLibraries}"PInterface.o ∂
"{PLibraries}"PasLib.o ∂
```

```
"{Libraries}"Runtime.o ∂
-o Creation ∂
-la > Creation.map
```

This example links the main application file Creation.p.o with the PInterface.o, PasLib.o, and Runtime.o libraries, placing the output in an application file named Creation and writing a linker map to the Creation.map file. Creation is an application that can be launched from the Finder or executed from MPW.

This is a more sophisticated example:

```
Link -rt MROM=8 -c 'MPS ' -t ZROM -ss 140000 ∂
 -l > TheROMListing -o TheROMImage {LinkList}
```

This command links the files defined in the Shell variable {LinkList} into a ROM image file, placing the output in the TheROMImage file. The segment size is set to 140,000 bytes, and the ROM is created as a resource 'MROM' with ID=8. The file is typed as being created by MPW (creator 'MPS '), with file type ZROM. Link's location-ordered listing is placed in the TheROMListing file.

▶ Creating an Object-Code Library

When you are writing a large application, and the length of time that it takes to compile and link the program starts to annoy you, it's time to start thinking about creating object-code libraries: chunks of code that have been precompiled and therefore don't have to be compiled from scratch every time you build your program.

When you write a program using MPW, you can convert any portion of it into a precompiled library by using the MPW Lib tool, which is a part of the basic MPW development system.

▶ The Lib Command

You can invoke MPW's Lib tool by using the Lib (rhymes with "vibe") command. Once you have created a library by using the Lib command, you can include your library in a makefile. Then, by using the commands CreateMake and Make (either directly or from the MPW Build menu), you can link any routine in the main body of the program with any routine in your library, without having to recompile the library every time you build the program.

The syntax of the Lib command is:

```
Lib [option...] objectFile...
```

The Lib command combines the files specified in the objectFile parameter into a single object file. Input files must have the file type 'OBJ '. Lib reorganizes the input files, placing the combined libary file in the data fork of the output library file. By default, the output library file is assigned type 'OBJ ' and the creator 'MPS '.

This is an example of a Lib command:

```
Lib {CLibraries} -o {CLibraries}CLibrary.o
```

This command combines all the library object files in the {CLibraries} directory into a single library named CLibrary.o. For applications that require most or all of the C library files, using the new CLibrary file can sometimes reduce link time.

The format for including a library file in a makefile is:

```
Libs = "{Libraries}"Interface.o
```

▶ What To Put in a Library

Once you have started using object-code libraries, it isn't difficult to figure out when the time has come to convert a portion of a program into a library. When you have compiled and debugged a block of code and haven't made any changes in it for a while, chances are that it has become a good candidate for conversion into a library. You can turn it into a library with the Lib command, include the library in your program's makefile, and then build your program at any time you like without having to wait for your library to be recompiled. That can save you a lot of time every time you build your program.

If you decide that you want to change a piece of code that you have tucked away in a library, you can simply edit the library's source code and then recompile the library by using the Lib command again.

Note ▶

Another Time-Saver. Another way to speed up the build process is to increase your RAM cache. Large links run up to four times faster when you use a RAM cache of 64K or more on a computer with at least 1MB of RAM. You can increase your RAM cache from your computer's control panel. You must then restart your system for your increased RAM cache to take effect.

There are two more good reasons for using object-code libraries in MPW programs:

- Once you have written a library of routines that perform a certain function—for example, a block of code that sets up a window environment—you can use your library in any program you write by simply including the library in the program's makefile. You can thus build up a collection of libraries that can be used in various programs.
- You can combine object code from different files and languages into a single object file. For example, you can include blocks of code that were originally written in Pascal or assembly language in a program written in C. In fact, the creators of MPW used the Lib tool for just this purpose when they constructed the various libraries that come with the MPW system.

▶ Uses for the Lib Command

You can use the Lib tool to

- Convert a portion of a program into a library to reduce linking time and simplify program development.
- Combine object code from different languages into a single library.
- Combine several libraries into a single library.
- Delete unneeded modules from a program (with the -dm option).
- Change the segmentation of a program (with the -sg and -sn options).
- Change the scope of a symbol from external to local (with the -dn option).

The last three options can be useful when you want to construct a specialized library for linking a particular program.

How Lib Works

The Lib command, like the Link command, concatenates its output files. It also offers optional renaming, resegmentation, and deletion operations, as well as the option of overriding an external name.

Lib does not combine modules into larger modules, nor does it resolve cross-module references. This limitation guarantees that the output of a link that uses the output of Lib is the same as that of a link that uses the "raw" object-code files produced by the MPW compilers and the MPW assembler.

The Lib tool automatically handles file-relative scoping conventions, such as nested procedures in Pascal, static functions in C, and ENTRY names in assembly languages. It never confuses references to an external symbol with references to a local symbol of the same name, even if the two symbols are in two files combined with Lib.

Options Used with the Lib Command

These options can be used with the Lib command.

Option	*Function*
-d	Suppresses duplicate definition warnings.
-df *deleteFile*	Deletes modules listed in the *deleteFile* file.
-dm *name*[,*name*]...	Deletes external modules and entry points.
-dn name[,*name*]...	Deletes external names, making them local.
-mf	Uses MultiFinder temporary memory if necessary.
-o *name*	Writes object file *name* (default Lib.Out.o).
-p	Writes progress information to diagnostics.
-sg *newSeg=old*[,*old*]...	Merges old segments into new segment.
-sn *oldSeg=newSeg*	Changes segment name *oldSeg* to *newSeg*.
-sym [Off] [*keyword*]	Omits symbolic information. Keywords that can be used with the *keyword* option are [,NoLabels], which omits label information; [,NoLines], which omits source line information; [,NoTypes], which omits type information, and [,NoVars], which omits variable information.
-sym [On | Full]	Keeps symbolic information (default).
-ver *n*	Sets OMF file version number to *n*.
-w	Suppresses warning messages.

Status codes returned by the Lib command are as follows.

Status Code	*Meaning*
0	No error detected.
1	Syntax error.
2	Fatal error.

▶ Building a Program

Once you know how to build a program, you can simplify the process of compiling and linking the program by creating a makefile, and then executing the makefile using the Make command.

There are two ways to create a makefile: by writing it yourself, or by executing the MPW command CreateMake. You can issue the CreateMake command from a command line, a Commando dialog, or the MPW Build menu.

CreateMake can create a simple makefile script for any program written under MPW. The makefile generated by CreateMake contains a set of rules needed to compile and link the program. Once a makefile has been created, you can generate a set of commands to build the program by executing Make using the program's makefile script.

Figure 8-2 is a makefile that is used to build a sample program named Creation, which is listed in Pascal source code, in Appendix C.

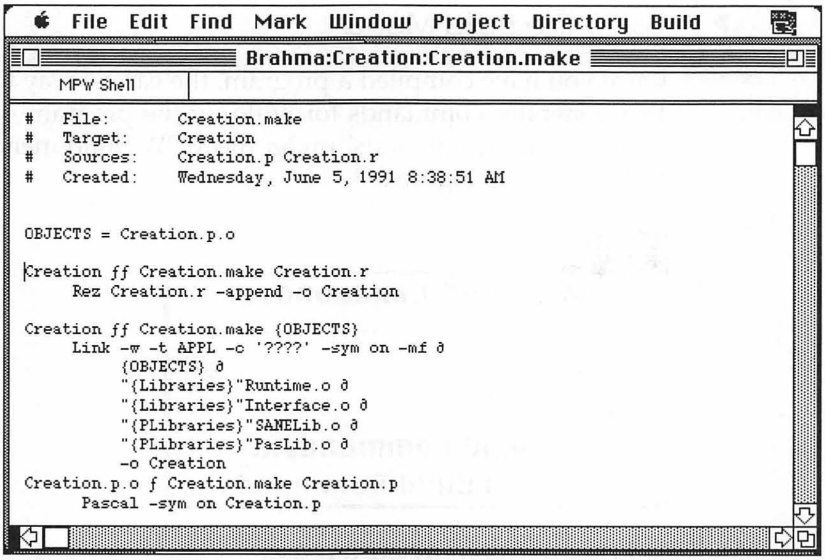


Figure 8-2. A makefile

It's easier to create a makefile using the CreateMake command than it is to write a makefile from scratch. However, a script created with CreateMake contains only the commands that are needed to compile and link the program: the commands Pascal, C, Asm, Rez, and Link, listed in whatever combination is needed to build the file properly. Collectively, these five commands are called build commands.

When you write a makefile script, instead of letting CreateMake write it for you, you don't have to settle for just five build commands; you can put as many commands in your makefile script as you like. For example, you can include commands to redirect the output of error messages, commands to copy files to other directories, and commands to print out maps and other kinds of listings that can be generated during the build process.

A disadvantage in using CreateMake is that it gives you little control over the many options that can be used with MPW's build commands. When you write your own makefile, you can select the options that you want to use with commands such as Pascal, C, Asm, Rez, and Link.

Fortunately, there is a way to take advantage of the work-saving features offered by CreateMake without having to live with its limitations. You can create a makefile using CreateMake and then customize it by adding your own options and commands. Of course, to do that, you must know how a makefile works. But that will be no problem by the time you finish this chapter. The architecture and operations of makefiles are described later under the heading "Writing a Makefile."

▶ Using the Build Menu

Once you have compiled a program, the easiest way to create a makefile that generates commands for building the program is to select the item "Create Build Commands" under the MPW Build menu. The Build menu is illustrated in Figure 8-3.

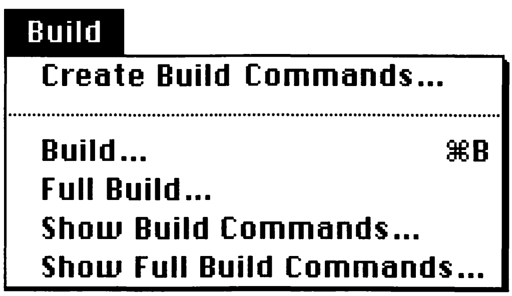


Figure 8-3. The MPW Build menu

When you select the menu item "Create Build Commands," MPW displays a Commando dialog like the one shown in Figure 8-4. By simply clicking on controls in the CreateMake Commando, you can create a makefile that can help you build your program.

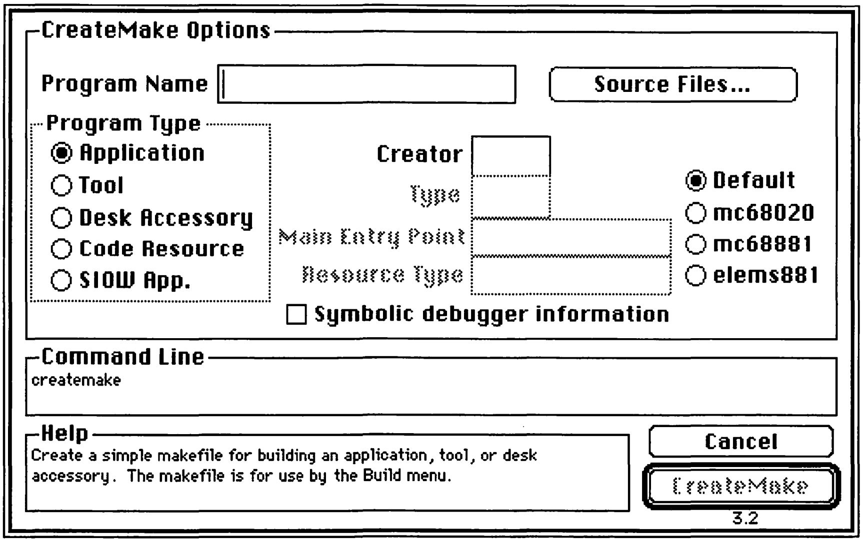


Figure 8-4. CreateMake Commando

You can also invoke the CreateMake Commando by executing the command

```
Commando CreateMake
```

or the command

```
CreateMake…
```

(making sure, of course, that you generate the ellipsis in the second example by typing Option-Semicolon, not by typing three periods).

If you want to execute CreateMake by typing a command line rather than using a Commando dialog, you can execute the command from a command line.

▶ The CreateMake Command

The syntax of the CreateMake command is:

```
CreateMake [ -Application [ -c creator ] | -Tool | -DA
| -CR -m mainEntryPoint -rt resourceType [ -t type ]
[ -c creator ] ] [-sym on] [ -mc68020 | -mc68881 |
-elems881 ] ProgramFile...
```

The CreateMake command creates a makefile script, a special kind of MPW script that can be used to build a program. The parameter ProgramFile is the name of the program to be built by the script. The makefile script that is created is named ProgramFile.make.

Note ▶

The BuildCommands and BuildProgram Commands. You can also create a makefile using two other commands: BuildCommands and BuildProgram.

BuildCommands works much like CreateMake; it generates and displays the commands needed to build a program, and you can then save those commands as a makefile script. BuildProgram also generates and displays the commands needed to build a file. Then it goes one step farther and actually builds the program.

BuildCommands and BuildProgram are used far less frequently than CreateMake; for more information about them, see the *MPW 3.0 Reference*.

When you create a makefile using the CreateMake command, you must pass the command a list of files to be included in the makefile. This list can include both source and library files.

Source files included in a CreateMake command must have the suffix .a (for assembly language files), .c (for C language files), .p (for Pascal files), .cp (for C++ files), or .r (for resource description files). You can include library files by using the suffix .o.

It is not necessary to type the names of the MPW libraries listed in Table 8-1. CreateMake finds any MPW libraries that are needed to create the makefile and includes them automatically in makefile script.

When you create a makefile using CreateMake, you can use choose the kind of program that you want the makefile to build. If you want a makefile that builds an application, you do not have to use the

-Application option because a script to build an application is the default. Other options are -Tool (to create an MPW tool makefile); -DA (a desk accessory makefile); -CR (for a makefile that builds a standalone code resource); and -rt (for a makefile that builds an ordinary resource).

CreateMake does not place references to #include files or USES files in the makefiles that it creates. Libraries other than those listed in Table 8-1 are not included in makefiles generated by CreateMake unless they are specified as parameters.

This is an example of a CreateMake command:

```
CreateMake -tool Create Create.c Create.r
```

This command creates the makefile shown in Figure 8-2 and listed at the end of this chapter.

▶ Options Used with the CreateMake Command

These options can be used with the CreateMake command:

Option	*Function*
-Application	Creates an Application (default).
-c creator	Assigns a creator name (optional; for makefiles that create applications or stand-alone resources).
-CR	Creates a code resource.
-DA	Creates a desk accessory.
-elems881	Generates 68881 instructions for transcendental functions.
-Tool	Creates an MPW tool.
-m mainEntryPoint	Main entry point (required for code-resource makefiles).
-mc68020	Generates 68020 instructions.
-mc68881	Generates 68881 instructions for elementary operations.
-rt resourceType	Resource type (required for code-resource makefiles).
-SIOW	Creates a Simple Input/Output Window.
-t type	File type (required for code-resource makefiles).
-sym on	Includes SADE debugging information in the object file.

Status codes returned by the CreateMake command are as follows.

Status Code	*Meaning*
0	No errors encountered.
1	Parameter or option error.

▶ Writing a Makefile

If you like to type, you can write your own makefiles instead of using the CreateMake command. To do that, you'll need to be familiar with the information covered in this section. Alternatively, you can use the material in this section to customize makefiles that have been created using the CreateMake command.

Once you have created a makefile that contains the rules for building a program, you can use the CreateMake command to expand the rules in your makefile into a set of commands to build the program.

▶ The Make Language

The most important thing to understand about makefiles is that a makefile script is written in a combination of two languages: the MPW command language and a special Make language.

In a makefile script, you can use any MPW command. But a typical makefile also contains a series of special commands called dependency rules.

One difference between a makefile and an ordinary script is that you can define a variable in a makefile by using the = operator. For example, you could create a variable in a makefile script by including the following line in the script.

```
Libs   =   "{Libraries}"Interface.o ∂
           "{Libraries}"Runtime.o ∂
           "{Libraries}"PasLib.o
```

Notice that the line-continuation character ∂ (Control-D) can be used in a makefile script, in the same way it is used in any other kind of script written in the MPW command language.

This command declares a variable called {Libs} and defines its value as the string "{Libraries}"Interface.o "{Libraries}"Runtime.o "{Libraries}"PasLib.o. Later in the makefile, the {Libs} variable can be used in place of the string that is its value.

▶ The *ƒ* and *ƒƒ* Operators

Another unique characteristic of a makefile script is that it can contain two operators made up of special characters. One of those operators is the *ƒ* character (Option-F). The other operator is two Option-Fs: *ƒƒ*. The *ƒ* and *ƒƒ* operators are found only in makefiles, never in any other kind of MPW script.

In a makefile, a command containing the *ƒ* operator or the *ƒƒ* operator is known as a dependency rule. A dependency rule always includes two lines: a line containing the *ƒ* operator or the *ƒƒ* operator, and a second line containing a build command.

The first line of a dependency rule—the line containing *ƒ* or *ƒƒ*—is called a dependency line. The second line of a dependency rule is a build command line. The first line of a dependency line—that is, its dependency rule—is always typed flush left. The second line of the dependency line—that is, its build command line—is always indented, using a tab or (less commonly) one or more spaces.

▶ The Single-*ƒ* Dependency Rule

This is the format of a dependency line containing the *ƒ* operator:

```
targetFile ƒ prerequisiteFile...
```

In a dependency line, the *ƒ* operator means "depends on," or "is a function of." Thus, the targetFile in the preceding example "depends on," or "is a function of," the prerequisite files. That means that the target file is rebuilt only under one of two conditions: if it does not exist, or if the prerequisite file is newer than the target file.

A dependency line can include more than one prerequisite file. If there are two or more prerequisite files, they are compiled in the order in which they appear in the dependency line.

Dependency rules are used in makefiles to prevent MPW from building files when compilation is not required. If a target file does not exist, it is built. If a prerequisite file associated with a target file has changed since the last time the target file was built, the target file is rebuilt.

If a target file exists, and if the prerequisite file associated with the target file has not changed since the last time the target file was compiled, then the target file is not compiled, because that would be a waste of time.

This example shows how the operator ƒ can be used in a makefile, along with an associated build command:

```
Creation.p.o ƒ Creation.p
    Pascal -sym on Creation.p
```

In the dependency line that appears on the first line of this example, Creation .p.o is the target file and Creation.p is the prerequisite file. Thus, if the file Creation.p.o does not exist, it is compiled. Creation.p.o is also compiled if the file Creation.p is newer than Creation.p.o.

The indented line that follows the dependency line contains the actual build command associated with the files specified in the dependency rule. Since Creation.p is a Pascal file (we know that because its name ends with the suffix ".p"), it is compiled using the build command

```
    Pascal -sym on Creation.p
```

Thus, the second line of the two-line example is the line that compiles Creation.p.o if compilation is required. In this case, the file Creation.p is compiled using the option -sym on, which means that symbols which are needed by the SADE debugger are included in the compiled file.

To compile a C source file named Creation.c, you could use the dependency rule

```
Creation.c.o ƒ Creation.c
    Pascal -sym on Creation.c
```

Similarly, you could assemble an assembly language source file named Creation.a by using the dependency rule

```
Creation.a.o ƒ Creation.a
    Asm -sym on Creation.a
```

In this case, the Creation.p.o file depends on both Creation.p and Menu.p. Thus, the Creation.p.o file is compiled if it does not exist, or if either Creation.p or Menu.p is newer than Creation.p.o.

Since a program must be linked after it is compiled, a typical makefile includes Link commands as well as compilation commands. Thus a makefile that builds the Creation file could include both of these dependency rules:

```
Libs =   "{Libraries}"Interface.o ∂
         "{Libraries}"Runtime.o ∂
         "{Libraries}"PasLib.o
Creation.p.o ƒ Creation.p
     Pascal ƒ -sym on Creation.p
Creation ƒ Creation.p.o
     Link -o Creation Creation.p.o {Libs}
```

This example begins with a variable definition that was presented earlier in this chapter. In this definition, a variable called Libs is declared, and its value is defined as the string "{Libraries}"Interface.o "{Libraries}"Runtime.o "{Libraries}"PasLib.o.

The second command in the example is a dependency rule which we have already examined; it compiles the file Creation.p.o if it does not exist, or if the file Creation.p is newer than Creation.p.o.

The second dependency rule is a new one; it links the Creation.p.o file (which is compiled by the first dependency rule) with the {Libs} variable. Because the -o option is used and is followed by file name Creation, Link writes its output to a file named Creation. That is the name of the application that the makefile builds.

▶ The Double-*ƒ* Dependency Rule

In some cases, a target file specified in a dependency line may depend on more than one prerequisite file, and the target file's prerequisite files may use different build commands. Suppose, for example, that the target file Creation.p.o depended on both the Pascal file Creation.p and the resource description file Creation.r.

As explained earlier in this chapter and in Chapter 6, Pascal source files are compiled using the Pascal command, and resource description files are compiled using the Rez command. In a case such this—namely, when a target file has multiple dependency paths, and each dependency path uses a different build command—the *ƒƒ* dependency operator is used, instead of the *ƒ* dependency operator.

To put this another way, you must use the *ƒƒ* operator in a dependency line instead of the *ƒ* when:

- a target file has more than one prerequisite file, and
- some prerequisite files use different build commands.

The syntax of a dependency line containing the *ƒƒ* operator is the same as the syntax of a dependency line containing the *ƒ* operator:

targetFile ƒƒ prerequisiteFile...

The *ƒƒ* operator, like the *ƒ* operator, means "depends on." Thus the target file specified in the preceding example "depends on" the specified prerequisite file. That means that the target file is compiled only under one of two conditions: if it does not exist or if the *prerequisiteFile* is newer than the target file.

Listing 8-1 is a complete makefile that contains a variable definition and both single-*ƒ* and double-*ƒ* dependency rules. It was created with the MPW Build menu for the Creation.p program in Appendix C. Notice that comments in a makefile are preceded by the # symbol, just as in any other kind of script written using the MPW command language.

Listing 8-1. A sample makefile

```
Libs    =   "{Libraries}"Interface.o ∂
            "{Libraries}"Runtime.o ∂
            "{Libraries}"PasLib.o
Creation     ƒƒ     Creation.r         # Creation depends on Creation.r
      Rez Creation.r -a -o Creation    # Creation.r's Rez command
Creation     ƒƒ     Creation.p.o       # Creation depends on Creation.p.o
                    Creation.r         # ... and Creation.r
      Link -o Creation ∂               # Creation's Link command
      Creation.p.o {Libs}
Creation.p.o ƒ Creation.p              # Creation.p.o depends on
Creation.p
      Pascal ƒ -sym on Creation.p      # ...Creation.p's Pascal command
Creation ƒ Creation.p.o
```

In Listing 8-1, the *ƒƒ* operator is used because the application file Creation depends on both Creation.r and Creation.p.o, and because Creation.r and Creation.p.o use different build commands.

Notice that the Creation.r file appears in both double-*ƒ* dependency rules shown in Listing 8-1, and that the Creation.p.o file also appears in two dependency rules: the first double-*ƒ* rule, and in the last rule in the listing—the rule in which Creation.p.o is compiled.

This is how the makefile shown in Listing 8-1 works:

1. In accordance with the last dependency rule in the listing, the Creation.p.o file is compiled using the Creation.p file if the Creation.p.o file does not exist or if the Creation.p file is newer than the Creation.p.o.file.

2. In accordance with the first double-*ƒ* rule in the listing, the Creation file is compiled using the Creation.r file if the Creation file does not exist, or if the Creation.r file is newer than the Creation file.
3. In accordance with the second double-*ƒ* rule in the listing, the Creation file is linked using both the Creation.p.o and Creation.r files if the Creation.r file is newer than the Creation file. (When this rule is invoked, there will be no situation in which the Creation file does not exist because if it does not exist, it will be compiled by the first double-*ƒ* dependency rule.)
4. If the second double-*ƒ* rule causes the Creation file to be linked, the files used in the link are the Creation.p.o file and the libraries that equate to the {Libs} variable. If the link occurs, the application file that is output is named Creation.

If you take the time to analyze this process, you will see that the makefile shown in Listing 8-1 performs only the compilations and links that are necessary to build an up-to-date version of the Creation application file.

▶ Makefiles in a Nutshell

To sum up, this is how makefiles work:

- A makefile is a text file that describes dependency information for one or more target files. A target file is a file to be built or rebuilt; it depends on one or more prerequisite files that must exist or be brought up to date before the target file can be built or rebuilt. For example, an application typically depends on its source file or files, its resource file or files, and several library files. If any of a target file's prerequisite files are newer than the target file, the makefile rebuilds the target file.
- A target file's prerequisite files may themselves be target files with their own prerequisite files. This process cascades; prerequisite files that are also target files can have their prerequisite files, and so on.
- A makefile can contain dependency rules, variable definitions, and comments.

Note ▶

Tips for Writing Makefiles. These are some additional facts that may prove useful when you write or customize makefiles:

- A makefile's physical input lines can be no more than 255 lines long. However, logical input lines (lines made up of more than one physical line continued with the character ∂) may be of any length.
- Makefiles use the same quoting conventions as do other kinds of scripts written in the MPW command language. Single quotation marks can be used to delimit a string that is to be interpreted literally, and double quotation marks can be used to quote strings in which variable references are expanded and the ∂ character (Option-D) is recognized as an escape character.
- You can use shell variables in makefiles, and you can define your own variables using the = operator.

▶ The Make Command

When you have created a makefile for a specified target file, you can process the makefile—thus creating a list of commands that are needed to build the target program—by issuing the MPW command Make.

▶ Building a Program with the Make Command

The Make command, when supplied with the appropriate information, outputs the commands needed to build a program. You can then build the program in one of three ways:

1. You can use the mouse to select the commands generated by the Make command, and then press the Enter key to execute them.
2. You can copy the commands generated by the Make command into an MPW script.
3. If you have created your makefile by selecting "Create Build Commands" from the MPW Build menu, you can build and execute the program by selecting either the "Build" or "Full Build" item from the MPW menu, as explained later in this chapter.

Once you have built a program in this fashion, you can execute it from the Finder, or MPW.

The syntax of the Make command is:

```
Make [option...] [target...]
```

Make does its work by reading a makefile: a text file that describes the dependencies of the various components of a program, and that lists the shell commands needed to rebuild the target. You can specify the makefile that you want Make to read by using the -f option. If you wish, you can specify more than one makefile to be processed by the Make command.

If you execute the Make command without specifying a target file, the target on the left side of the first dependency rule in the makefile is built.

This is an example of a Make command:

```
Make -p -f Create.make Creation
```

This command builds the target file Creation. Because the -p option is used, Make writes progress information to standard output, usually the screen. The -f Create.make option instructs the Make command to generate its build commands by reading a makefile called Create.make. Dependency rules used to create the target file Create are read from the Create.make makefile.

After Make processes a makefile, it generates the commands that are needed to build or rebuild the target file. By default, these commands are written to standard output. However, you can execute the build commands that Make generates after they are generated.

The Make command processes a makefile (or makefiles) in two phases:

1. Make reads the specified makefile (or makefiles) and creates a dependency graph for the target file.
2. Make generates build commands for the target file. If the target depends on prerequisite files that are out of date, Make generates command lines for updating the target file. Build commands are issued first for lower level dependencies that need to be rebuilt and then for higher level dependencies.

Options Used with the Make Command

These options can be used with the Make command:

Option	*Function*
-d *n*[=*v*]	Defines a variable *n* with the value *v* (overrides variable definitions included in the makefile).
-e	Rebuilds everything that is a part of the specified or default target, regardless of whether the target is out of date (overrides normal dependency rules).
-f *m*	Reads dependencies from makefile *m* (default is any file named MakeFile in the current directory).
-p	Writes progress information to diagnostic output.
-r[*target*]	If no *target* is specified, this option causes Make to find all the roots (top-level targets) of the dependency graph created by Make. If a *target* is specified, Make finds the root (or roots) for which the target is a prerequisite. You can instruct Make to write out this information by using the -s option. The -r option overrides normal processing of the Make command; it is used in debugging and for analyzing complex makefiles.
-s	Shows structure of dependencies. This option writes a dependency graph for the specified targets to standard output. The -s option overrides normal processing of the Make command; it is used in debugging and for analyzing complex makefiles.
-t	"Touches" dates of targets and prerequisites; that is, brings files up to date by adjusting their modification dates, without generating build commands.
-u	Writes to diagnostic output a list of "unreachable" targets—that is, targets that are not prerequisites (or prerequisites of prerequisites) of the specified target.
-v	Writes verbose (detailed) error and progress information to diagnostic output.
-w	Suppresses warning messages.

Status codes returned by the Make command are as follows.

Status Code	*Meaning*
0	Successful completion.
1	Parameter or option error.
2	Execution error.

▶ The Build Menu

The Make command does not actually build a program; it merely displays the commands needed to build the program. You can then build the program in one of three ways:

1. You can use the mouse to select the commands generated by the Make command, and then press the enter key to execute them.
2. You can copy the commands generated by the Make command into an MPW script, and then execute the script.
3. If you have created your makefile by selecting "Create Build Commands" from the MPW Build menu, you can build and execute the program by selecting either the "Build" or "Full Build" item from the MPW menu.

The easiest and most convenient way to build a program written under MPW is to use the MPW Build menu. The Build menu has four items:

- The first menu item, "Create Build Commands," displays the Commando dialog for the MPW command CreateMake. You can then execute the CreateMake Commando to create a makefile containing the commands needed to build a program. When you create a makefile in this way, MPW names the makefile *program*.make (where program is the name of the target program). MPW then reads that makefile each time the Make command is executed. (If there is no file named *program*.make, MPW looks in the current directory for a file named MakeFile and reads that.)
- When you have created a makefile for a program using the menu item "Create Build Commands," you can build the program by selecting the second item on the Build menu, also named "Build." When you select the Build menu item, MPW displays a dialog asking you the name of the file to be built. MPW then builds the specified program using the dependency rules specified in the program's makefile.
- The third menu item, "Full Build," works just like the Build menu item. But "Full Build" does a complete build of the specified program, rebuilding all its components, regardless of the makefile's dependency rules.
- The fourth menu item, "Show Build Commands," displays a dialog that asks you to specify a program. It then displays the commands currently needed to build the program.

- The fifth and last menu item, "Show Full Build Commands," works just like "Show Build Commands." But "Show Full Build Commands" displays all the commands that are needed to build the program if its makefile's dependency rules are ignored.

Once you have built a program using the MPW menu, you can execute it from the Finder, or MPW.

Creation: A Sample MPW Program

Appendix C is a source-code listing of a sample Pascal program called Creation. Appendix D is the program's resource description file.

A makefile that generates the build commands for the program is listed in Appendix E.

The program is named Creation because you can use it as a template to create your own programs in MPW Pascal. It creates a window in which you can type text in any font and any style.

Creation is a simpler program than the TESample program that is packaged with the MPW Pascal and MPW C compilers. Although Creation takes advantage of the advanced text-styling capabilities now offered by TextEdit, it does not create a multi-window environment, and it does not create scroll bars—two features that make the TESample program quite complicated and therefore not very easy to understand.

Because Creation lacks these features (as important as they are), it is much easier to understand than the TESample program is. It could thus be considered a kind of prerequisite for studying the much more complex TESample program. If you type, compile, build, and execute Creation—and study it to find out how it works—you'll be well prepared to move on to Apple's more complex TESample program.

Conclusion

The book you have just finished is a complete guide to the Macintosh Programmer's Workshop. Since MPW programs are not written in a vacuum, it also includes chapters on Macintosh architecture and on the three Macintosh managers that give programmers the most trouble: the Event Manager, the Resource Manager, and the Memory Manager.

I hope that you have derived as much benefit from reading this book as I got from writing it—and I wish you much success in programming with MPW, the professional Macintosh programming environment.

Appendix A
The MPW Command Set

AddMenu — Add a menu item

```
AddMenu [menuName [itemName [command...]]]
```

- Add an item to the MPW menu bar.

Adjust — Adjust lines

```
Adjust [-c count] [-l spaces] selection [window]
```

- Shift all lines in a selection to the right by one or *count* tabs or *spaces*.

-c *count*	Adjust *count* times.
-l *spaces*	Shift lines by *spaces* spaces.

Alert — Display alert box

```
Alert [-s] [message...] < file
```

- Display an alert dialog.

-s	Silent: Don't beep when dialog is displayed.

Alias Define or write command aliases

```
Alias [name [word…]]  > aliasList
```

- Define *name* as an alias for MPW command *word*. If only *name* is specified, any alias definition associated with *name* is written to standard output. If both *name* and *word* are omitted, a list of all aliases and their values is written to standard or specified output.

Align Align text to left margin

```
Align [-c count] selection [window]
```

- Place all lines in *selection* the same distance from left margin as the first line.

-c *count*	Repeat the Align *count* times.

Asm Invoke MC68xxx Macro Assembler

```
Asm [option…] [file…]  < file > listing ≥ progress
```

- Assemble *file*.

-addrsize *size*	Set size of address display.
-blksize *blocks*	Use *blocks* * 512-byte I/O buffers.
-case on	Distinguish between upper- and lowercase.
-case obj[ect]	Preserve case in object file.
-case off	Ignore case (default).
-c[heck]	Syntax check only; don't create object file.
-d[efine] *name*	Same as *name* EQU 1.
-d[efine] *name=value*	Same as *name* EQU *value*.
-d[efine] *&name*	Same as *&name* SET[AC] 1.
-d[efine] *&name=value*	Same as *&name* SET[AC] *value*.
-e[rrlog] file	Write errors and warnings to *file*.
-f	Suppress page ejects in listing.
-font *name*[,size]	Set listing font and size.
-h	Suppress page headers in listing.
-i *directory,…*	Search for includes in *directory,…*.

Asm	(continued)	
	-l	Write full listing to output.
	-lo *objname*	Write listing to file or directory *objname*.
	-o *objname*	Generate code in file or directory *objname*.
	-pagesize *l*[,*w*]	Set listing page length and width.
	-print *mode*	Same as PRINT *mode*.
	-p	Write progress information to diagnostics.
	-s	Write short listing to output.
	-sym off	Include SADE information in the object file.
	-sym on | full	Don't include SADE information in the object file.
	-t	Write time and total lines to diagnostics.
	-w	Suppress warnings.
	-wb	Suppress warnings on branch instructions.

Backup Folder file backup

`Backup [option…] -from folder -to folder [file…] > commands ≥ progress`

- Generate a shell script that can make backups of files.

-a	Copy all files in "from" and not in "to."
-alt	Alternate prompts for disk drives.
-c	Create "to" folders if they don't exist.
-check *checkopt* [,*checkopt*]...	Produce reports based on *checkopt*.
	checkopt parameters:
	from: Files in "from," not in "to."
	to: Files in " to," not in "from."
	allfroms: Files in "from" and not in "to," even if none.
	alltos: Files in "to," not in "from," even if none.
	folders: Folders in "from," not in "to."
	newer: ObjectS in "to" newer than those in "from."
-co *filename*	Redirect "-check" reports to *filename*.

Backup (continued)

-compare [only][,'*opts*']	Write Compare commands for out-of-date files.
-d	Write Delete commands for files in "to" not in "from."
-do [only][,'*command*']	Write the command string specified by *command* . . .
-e	Eject disk when done.
-from *folder* | *drive*	Specify source *folder* or *drive* (1 or 2).
-l	Write directory listing of "from" files.
-lastcmd '*command*'	Write the *command* string as the last command.
-level *n*	Restrict -a and -d options to files beyond level *n*.
-m	Multi-disk: More than one "from" or "to" disk.
-n	Show folder nesting by indenting commands.
-p	Write progress information to diagnostics.
-r	Recursively process nested folders.
-revert	Revert "to" files to their "from" state.
-since *date*[,*time*] | *fname*	Process only files since specified time.
-sync	Synchronize both source and destination folders.
-t *type*	Process only files of specified type.
-to *folder* | *drive*	Specify destination *folder* or *drive* (1 or 2).
-y	Suppress duplicate -y option.

Beep Generate tones

```
Beep [note [,duration [,level]]]…
```

- Generate a tone from the built-in Macintosh speaker.

duration is specified in sixtieths of a second (default is 15).

Sound *level* is a number from 0 through 255 (default is 128).

Begin Group commands

```
Begin
    command...
End
```

- Execute *commands* as a group.

Break Break from For or Loop

`Break [If expression]`

- Exit from a loop if *expression* is true.

Browser Display MPW Browser tool (MPW 3.2)

`Browser`

- Display MPW Editor's Browser window.

BuildCommands Show build commands

`BuildCommands program [option...] > commands`

- Write to standard output the commands needed to build a program.

option... Options for Make command.

BuildIndex Create a BTree index file

`BuildIndex datafile`

- Create a BTree index file named *datafile*.index for use by the Get tool.

BuildMenu Create the Build menu

`BuildMenu`

- Create the MPW Build menu.

BuildProgram Build the specified program

`BuildProgram program [option…] > log`

- Build *program*.

option…	Options for the Make command.

C Invoke C compiler

`C [option…] [file] < file > preprocessor ≥ progress`

- Compile *file*.

-b	Use string constants; generate PC-relative references.
-b2	Use -b option and overlay string constants.
-b3	Overlay string constants, but not PC-relative references.
-c	Syntax check only; don't create object file.
-d *name*	Same as #define *name* 1.
-d *name=string*	Same as #define *name string*.
-e	Write preprocessor results to output.
-e2	Implies -e option and strips comments.
-elems881	Generate MC68881 code for transcendentals.
-i *directory*	Search for includes in *directory*.
-m	Generate 32-bit references for data (less efficient code).
-mbg ch8	Use V2.0-compatible MacsBug symbols.
-mbg off	Don't place MacsBug symbols in code.
-mbg on | full	Use full MacsBug symbols.
-mbg <*n*>	Include MacsBug symbols truncated to length <*n*> .
-mc68020	Generate MC68020 code.
-mc68881	Generate MC68881 code for arithmetic operations.
-n	Turn pointer incompatibility errors into warnings.
-o *objname*	Generate code in file or directory *objname*.
-p	Write progress information to diagnostic.

C (continued)

-r	Warn on calling a function that has no definition.
-s *segment*	Generate code in *segment*.
-sym off	Include SADE information in the object file.
-sym on | full	Don't include SADE information in the object file.
-t	Write compilation time to diagnostic.
-u *name*	Same as #undef *name*.
-w	Suppress warnings.
-w2	Emit even more warnings.
-y *directory*	Create temporary files in *directory*.

Canon — Canonical spelling tool

`Canon [option…] dictionary [file…] < file > new`

- Copy specified files to standard outputs, replacing identifiers with spellings in dictionary file.

-s	Use case-sensitive matching.
-a	Assembler identifiers (include $, %, @).
-c *n*	Consider only the first *n* characters.

Catenate — Concatenate files

`Catenate [file1…]  < file2> concatenation`

- Concatenate *file1* and *file2*.

CFront — C++ to C translator

- (MPW C++ precompiler is available as a separate product.)

CheckIn

Check a file into a project

```
CheckIn -w | -close | ([option…] files…) > progress
```

- Check *files* into a project (used with Projector).

-a	Check in all files in current directory.
-b	Check in files… as branches.
-c	Cancel if conflict occurs (avoids dialog).
-cf *file*	Put description of changes in *file*'s 'ckid' resource.
-close	Close the Check In window.
-cs *comment*	Include description of changes in *file*'s 'ckid' resource.
-delete	Delete the file after checking it in.
-m	Check out the files for modification after checking in.
-n	Answer no to all dialogs (avoids dialogs).
-new	Add a new file to the project.
-p	Write progress information to standard output.
-project *project*	Make *project* the current project.
-t *task*	Put *task* in *file*'s 'ckid' resource.
-touch	Touch the mod date of file after checking it in.
-u *user*	Name of current user (overrides {User} shell variable).
-w	Open the Check In window.
-y	Answer yes to all dialogs (avoids dialogs).

CheckOut

Check a file out from a project

```
CheckOut -w | -close | ([options…] files…) > progress
```

- Check *files* out of a project (used with Projector).

-a	Check out all the files in the current project.
-b	Check out specified files on a new branch.
-c	Cancel if conflict occurs (avoids dialog).
-cancel	Cancel the checkout of the files.

CheckOut (continued)

-cf *file*	Put description of changes in *file*'s 'ckid' resource.
-close	Close the Check Out window.
-cs *comment*	Include description of changes in *file's* 'ckid' resource.
-d *dir*	Directory where the checked-out files should go.
-m	Check out a modifiable copy of the file.
-n	Answer no to all dialogs (avoids dialogs).
-newer	Check out latest copy of all files in the project.
-noTouch	Don't touch the mod date of the checked out files.
-open	Open the files after checking out.
-p	Write progress information to standard output.
-project *project*	Name of project that contains the files.
-r	Recursively check out files.
-t *task*	A short description of task accomplished.
-u *user*	Name of current user.
-update	Check out latest copy of all files you already have.
-w	Open the Check Out window.
-y	Answer yes to all dialogs (avoids dialogs).

CheckOutDir Specify the directory where checked out files will be placed

```
CheckOutDir [-project project | -m] [-r] [-x |
directory]
```

- Place checked-out Projector files in *directory*.

-project *project*	Name of project to associate with the checkout directory.
-m	List the checkout directories of all root projects.
-r	Recursively set or display the checkout directories.
-x	Reset the checkout directories to ":".

Choose

Choose or list network file server volumes and printers

```
Choose [option...] [[zone]:server[:volume] ...]
```

- Interactively mount or list specified Appleshare volumes or printers.

-c	Output in the form of further Choose commands.
-cp	Print driver name and type of current printer.
-dr *driverFileName*	Name of printer driver file in system folder.
-guest	Log in to the file server as a guest.
-list	List entities (don't choose them).
-p	Print version information.
-pr	Choose printers (instead of file servers).
-pw *password*	Specify server log-in *password*.
-u *username*	Specify user name for server log-in.
-v	Verbose (print names of volumes really mounted).
-vp *volumePassword*	Specify volume password (to mount it).
-type *type*	Specify type of entity to list or choose (or ≈).

Clear

Clear the selection

```
Clear [-c count] selection [window]
```

- Clear *selection* from *window*.

-c *count*	Repeat the clearing operation *count* times.

Close

Close specified windows

```
Close [-y | -n | -c] [ -a | window...]
```

- Close *window*.

-y	Save modified windows before closing (avoids dialog).
-n	Don't save any modified windows (avoids dialog).
-c	Cancel if there is a modified window (avoids dialog).
-a	Close all the windows.

Commando Display a dialog interface for commands

`Commando [`*`command`*`] [`*`-modify`*`]`

- Display Commando dialog for *command*.

-modify	Enable Commando's built-in editor.

Compare Compare text files

`Compare [`*`option...`*`]` *`file1`* `[`*`file2`*`]  <` *`file2`* `>` *`differences`* `≥` *`progress`*

- Compare *file1* and *file2*.

-b	Treat several blanks or tabs as a single blank.
-c c1-c2[,c1-c2]	Compare only specified columns.
-d *depth*	Maximum stack depth.
-e *context*	Display specified number of context lines.
-g *groupingFactor*	Specifies minimum number of lines that must match.
-h *width*	Write differences in horizontal format.
-l	Do case-insensitive match (ignore case differences).
-m	Suppress displays of mismatched lines.
-n	Don't write to output if files match.
-p	Write progress information to diagnostics.
-s	Use static (fixed) grouping factor (see -g option).
-t	Ignore trailing blanks.
-v	Suppress line numbers in vertical displays.
-x	Don't expand tabs.

CompareFiles Compare text files and interactively view differences

`CompareFiles [-9 | -13 | -b x y | -Portrait | -TwoPage]` *`oldFile newFile`*

- Compare *oldFile* and *newFile*.

CompareFiles (continued)

-9	Assume a screen size of 512 × 342.
-13	Assume a screen size of 640 × 480.
-b x y	Tile windows into the rectangle specified by x y.
-Portrait	Screen size for Apple Macintosh Portrait Display.
-TwoPage	Screen size for Apple Two-Page Monochrome Monitor.

CompareRevisions Compare two revisions of a file in a project

```
CompareRevisions file…
```

- Compare the revisions of *file* (used with Projector).

Confirm Display a confirmation dialog box

```
Confirm [-t] [message…] < file
```

- Display a confirmation dialog containing the message *message*.

-t	Put three buttons (Yes, No, Cancel) in dialog.

Continue Continue with next iteration of For or Loop

```
Continue [If expression]
```

- Continue with next iteration of loop if *expression* is true.

Copy Copy selection to Clipboard

```
Copy [-c count] selection [window]
```

- Copy *selection* to *window*.

-c *count*	Copy the *n*th selection, where *n* = *count*.

Count Count lines and characters

`Count [-l] [-c] [file…] < file > counts`

- Count lines or characters in *file*.

-l	Write only line counts.
-c	Write only character counts.

CPlus Script to compile C++ source

- (MPW C++ precompiler is available as a separate product.)

CreateMake Create a simple makefile

```
CreateMake [-Application [-c creator] | -Tool | -DA |
-CR

-m mainEntryPoint -rt resourceType [-t type] ] [-sym
on]

[-mc68020 | -mc68881 | -elems881] program file….
```

- Create a makefile for *program* using files specified in the *file…* parameter.

-Application	Create an application (default).
-c *creator*	Program's creator (optional).
-Tool	Create an MPW tool.
-DA	Create a desk accessory.
-CR	Create a code resource.
-m *mainEntryPoint*	Main entry point (required for code resource).
-rt *resourceType*	Resource type (required for code resource).
-t *type*	File type (optional for code resource).
-sym on	Include SADE information in the object file.
-mc68020	Generate 68020 instructions.
-mc68881	Generate 68881 instructions for elementary operations.
-elems881	Generate 68881 instructions for transcendental functions.

Cut Copy selection to Clipboard and delete it

```
Cut [-c count] selection [window]
```

- Cut *selection* from *window*.

-c *count*	Cut the next *count* selections.

Date Write the date and time

```
Date ([-a | -s] [-d | -t] [-c seconds]) | [-n] > date
```

- Write current date or date and time to standard or specified output.

-a	Write abbreviated date (format: Wed., Jul 24, 1991).
-s	Write short date (e.g., 7/24/91).
-d	Write date only.
-t	Write time only.
-c *seconds*	Write date corresponding to *seconds*.
-n	Write seconds since January 1, 1904.

Delete Delete files and directories

```
Delete [-y | -n | -c] [-i] [-p] name... ≥ progress
```

- Delete file or directory *name*.

-y	Delete directory contents (avoids dialog).
-n	Don't delete directory contents (avoids dialog).
-c	Cancel if a directory is to be deleted (avoids dialog).
-i	Ignore errors (no diagnostics).
-p	Write progress information to diagnostics.

DeleteMenu Delete user-defined menus and menu items

```
DeleteMenu [menuName [itemName]]
```

- Delete specified menu or menu item.

DeleteNames Delete user-defined symbolic names (used with Projector)

`DeleteNames [-u user] [-project project] [-public] [-r] [names… | -a]`

- Delete symbolic *names* used to represent a set of revisions under Projector.

-u *user*	Name of current user.
-project *project*	Name of project that contains the files.
-public	Delete public names.
-r	Delete names recursively.
-a	Delete all names.

DeleteRevisions Delete previous revisions of files in a project

`DeleteRevisions [-u user] [-project project] [-file] [-y] revision…`

- Delete previous revisions in Projector project *project*.

-u *user*	Name of current user.
-project *project*	Name of project that contains the files.
-file	Delete the file and all its revisions.
-y	Delete the file/revision (avoids dialog).

DeRez Resource decompiler

`DeRez [option…] resourceFile [file…] > description ≥ progress`

- Decompile *resourceFile*.

-c[ompatible]	Generate output compatible with Rez 1.0.
-d[efine] *name*[=*value*]	Same as #define *name* [*value*].
-e[scape]	Don't escape chars < $20 or > $D8.
-i[nclude] *pathname*	Search *pathname* when looking for #include files.
-m[axstringsize] *count*	Write strings *count* characters per line.
-only *typeExpr*	Process only resources of type *typeExpr*.
-p[rogress]	Write progress information to diagnostics.

DeRez (continued)

-rd	Suppress warnings for redeclared types.
-s[kip] *typeExpr*	Skip resources of type *typeExpr*.
-u[ndef] *name*	Same as #undef *name*.

Note: A *typeExpr* may have one of these forms:

```
type
"'type'(id)"
"'type'(id:id)"
"'type'(∂"name∂")"
```

Directory Set or write the default directory

`Directory [-q | directory] > directory`

- Set the directory to directory; if directory is not specified, write the name of the current directory to standard or specified output.

-q	Don't quote directories with special characters.

DirectoryMenu Create the Directory menu

`DirectoryMenu [directory…]`

- Create the MPW Directory menu; list directories specified in *directory* parameter.

DoIt Highlight and execute a series of shell commands

`DoIt (CommandFile [-echo] [-dump]) | [-selection]`

- Execute the commands in *CommandFile*.

-echo	Echo commands before execution.
-dump	Dump unexecuted commands after error.
-selection	Execute command in the current selection.

DumpCode Write formatted resources

`DumpCode [option...] resourceFile > dump ≥ progress`

- Disassemble object code stored in resources, and write formatted assembly code to standard or specified output.

-d	Don't dump object code.
-di	Suppress display of data initialization code.
-h	Don't write headers (offsets, hex, etc.).
-jt	Don't dump jump table.
-n	Dump only resource names.
-p	Write progress information to diagnostics.
-r *byte1*[,*byteN*]	Dump code from address *byte1* [through *byteN*].
-rt *type*[=*id*]	Dump only resources with this *type* [and *id*].
-s *name*	Dump only resource with this name.

DumpFile Display contents of any file

`DumpFile [option...] fileName > dump ≥ progress`

- Display the contents of *fileName*.

-a	Suppress display of ASCII character values.
-bf	Display both forks of the file.
-g *nn*	Group *nn* bytes together without intervening spaces.
-h	Suppress display of hexadecimal characters.
-o	Suppress display of file offsets.
-p	Write progress information to diagnostic output.
-r *byte1*[,*byteN*]	Display only the byte range from *byte1* to *byteN*.
-rf	Display the resource fork of the file (default is data fork).
-w *nn*	Display width of *nn* bytes on each line of output.

DumpObj

Write formatted object file

```
DumpObj [option…] objectFile  > dump ≥ progress
```

- Disassemble object code stored in the data fork of *objectFile*.

-d	Don't dump object code.
-h	Don't write headers (offsets, hex, etc.).
-i	Use ids, rather than names, in dump.
-jn	Just use names, rather than ids, in dump.
-l	Dump file locations of object records.
-m *name*	Dump only module *name*.
-mods	Dump a module summary with entry point information.
-mh	Omit module summary header.
-n	Dump only the dictionary of names.
-p	Write progress information to diagnostics.
-r *byte1*[,*byteN*]	Dump code from *byte1* in file [through *byteN*].
-sym [Off]	Disable symbolic output.
[On ǀ Full]	Enable symbolic output (default); can be followed by:
	[,NoLabels] Omit label information.
	[,NoLines] Omit source line information.
	[,NoTypes] Omit type information.
	[,NoVars] Omit variable information.

Duplicate

Duplicate files or directories

```
Duplicate [-y | -n | -c] [-p] [-d | -r] name… target
≥ progress
```

- Duplicate (copy) file or directory *name* to file or directory *target*.

-y	Overwrite target files (avoids dialog).
-n	Don't overwrite target files (avoids dialog).
-c	Cancel if conflict occurs (avoids dialog).
-p	Write progress information to diagnostics.
-d	Duplicate data fork only.
-r	Duplicate resource fork only.

Echo

Echo parameters

`Echo [-n] [parameter…] > parameters`

- Echo *parameter* to standard or selected output.

-n	Don't write return following the parameters.

Eject

Eject volumes

`Eject [-m] volume…`

- Eject *volume*.

-m	Leave the volume mounted.

Entab

Convert runs of spaces to tabs

`Entab [option…] [file…] < file > tabbed ≥ progress`

- Convert runs of spaces in *file* to tabs.

-a *minValue*	Minimum run of blanks that can be replaced with a tab.
-d *tabValue*	Input tab setting.
-l *quote…*	List of left quotes that prevent EnTab (default '").
-n	EnTab everything, including spaces inside quotes.
-p	Write progress information to diagnostics.
-q *quote…*	Quotes that prevent EnTab (default '").
-r *quote…*	List of right quotes that prevent EnTab (default '").
-t *tabValue*	Output tab setting.

Equal

Compare files and directories

`Equal [-d | -r] [-i] [-p] [-q] name... target > differences ≥ progress`

- Compare file or directory *name* with file or directory *target*.

-d	Compare data forks only.
-r	Compare resource forks only.
-i	Ignore files in target not in directory name.
-p	Write progress information to diagnostics.
-q	Quiet: Don't write output, just set {Status}.

Erase

Initialize volumes

`Erase [-y] [-s] volume...`

- Initialize *volume*.

-y	Yes; erase the disk (avoids dialog).
-s	Single-sided 400K disk (default is double-sided 800K disk).

Evaluate

Evaluate an expression

`Evaluate [-h | -o | -b] [word...] > value`

- Evaluate list of words *word...* and write result to standard or selected output.

or

`Evaluate name [binary operator] = expression.`

- Evaluate *expression* and assign the result to the variable *name*.

-h	Display result in hexadecimal (leading 0x).
-o	Display result in octal (leading 0).
-b	Display result in binary (leading 0b).

Execute

Execute command file in the current scope

`Execute commandFile`

- Execute *commandFile* in the current scope.

Exists Confirm the existence of a file or directory

`Exists [-d | -f | -w] [-q] name... > file`

- Confirm the existence of file or directory *name*.

-d	Check if name is a directory.
-f	Check if name is a file.
-w	Check if name is a file and writeable.
-q	Don't quote file names with special characters.

Exit Exit from a command file

`Exit [status] [If expression]`

- Terminate execution of script in which the Exit command appears if *expression* is true. Status of script is returned in *status* argument, if *status* argument is used.

Export Make variables available to commands

`Export [-r | -s | name...] > exports`

- Make variable *name* available to scripts and tools.

-r	Generate Unexport commands for all exported variables.
-s	Print the names only.

FileDiv Divide a file into several smaller files

`FileDiv [option...] file [prefix] ≥ progress`

- Divide *file* into smaller files.

-f	Split file at formfeed character.
-n *splitPoint*	Split file after *splitPoint* lines.
-p	Write progress information to diagnostics.

Files List files and directories

`Files [option…] [name…] > fileList`

- List contents of directory or volume *name*.

-c *creator*	List only files with this *creator*.
-d	List only directories.
-f	List full path names.
-i	Treat all arguments as files.
-l	Write files in long format (type, creator, size, dates, etc.).
-m *n*	Multi-column format, where *n* = columns.
-n	Don't print header in long or extended format.
-o	Omit directory headers.
-q	Don't quote file names that contain special characters.
-r	Recursively list subdirectories.
-s	Suppress the listing of directories.
-t *type*	List only files of this type.
-x *format*	Use extended format; fields specified by *format*. These characters can be used to specify the format:

a	Flag attributes.
b	Logical size, in bytes, of the data fork.
c	Creator of File ("Fldr" for folders).
d	Creation date.
g	Group (only for folders on a file server).
k	Physical size in kilobytes of both forks.
m	Modification date.
o	Owner (only for folders on a file server).
p	Privileges (only for folders on a file server).
r	Logical size, in bytes, of the resource fork.
t	Type.

Find Find and select a text pattern

`Find [-c count] selection [window]`

- Find *selection*.

-c *count*	Find the *n*th selection, where *n* = *count*.

Flush Flush tools that the shell has cached

`Flush`

- Flush the tools that the shell has cached.

For Repeat commands once per parameter

```
For name In word…
command….
End
```

- Execute *command* once for each word in the *word*… list.

Format Set or display formatting options for a window

```
Format [[-f fontName] [-s fontSize] [-t tabSize]
[-a attr]] | [-x fmt] [window…]
```

- Set the format of *window*. Default is the target window.

-f *fontName*	Set font to *fontName*.
-s *fontSize*	Set the font size to *fontSize*.
-t *tabSize*	Set the tab size to *tabSize*.
-a *attr*	Set auto indent to *attr*, and show invisibles flags. The *attr* parameter is a string made up of these characters: A Auto indentation on. a Auto indentation off. I Show invisibles on. i Show invisibles off.

Format (continued)

-x *fmt*	Specifies output format (*fmt*). The *fmt* parameter is a string made up of these characters: f Font name. s Font size. t Tab size. a Attributes.

Get Get a record from a file with a BTree index.

`Get [option…] datafile…`

- Get a record from file *datafile*.

-col *n*	Use *n*-column format (n = 1 through 6).
-d *default*	Use default keyword if no keyword is specified.
-h	Write header.
-k *keyword*	Use keyword in the datafile's index file.
-l	List all keys beginning with keyword.
-nf	No filtering; include field tags.
-q	(Quiet) No output when keyword not found.
-s	Use the selection in the active window as keyword.
-search	Text search datafile for occurrences of keyword.
-t	Write out template of the requested function/procedure.

GetErrorText Display error messages based on message number

`GetErrorText [-f filename] [-s filename] [-n] [-p] errNr[,insert,…] …`

- Display error message that corresponds to error number (*errNr*).

or

`GetErrorText -i idNr,…`

- Display error message that corresponds to the System Error Handler ID number *idNr*.

GetErrorText (continued)

-f *filename*	Display a tool's error message file name.
-s *filename*	Display error message file name for a system error.
-n	Suppress error numbers in displayed messages.
-p	Write SysErr's version info to diagnostics.
-i *idNr*	Report meaning of System Error Handler ID number.

GetFileName Display a Standard File dialog box

```
GetFileName [-q] [-s] [-c | [[-t type]… | -p | -d]
[-m message] [-b buttonTitle] [pathname]]
```

- Display a Standard File dialog window

-q	Suppress quoting of file names.
-s	Return 0 status even if cancel is clicked.
-c	Write current standard file path to standard output.
-t *type*	Specify file type for SFGetFile dialog.
-p	Display an SFPutFile dialog.
-d	Display an SFGetFile dialog for selecting a directory.
-m *message*	Specify a prompt message.
-b *buttonTitle*	Specify the default button's title.

GetListItem Display items for selection in a dialog box

```
GetListItem [option…] [[item…] | < file]
```

- Display a list dialog containing the specified items and read user input.

-c[ancel]	Return a status of 0 even when cancel is clicked.
-d[efault] *item*	Specified *item* is placed in list and is default selection.

GetListItem (continued)

-m[essage] *message*	Specified *message* is displayed in dialog above the list.
-q[uote]	Don't quote items in the output.
-r[ows] *n*	Display a list with *n* rows.
-s[ingle]	Allow user to make only one selection.
-w[idth] *width*	Make the list *width* pixels wide.

Help Write summary information

```
Help [-f helpfile] [command…]  > helpInformation
```

- Display MPW Help information.

-f *helpfile*	Use alternate help file (default is MPW.Help).

If Conditional command execution

```
If expression
   command…
[Else If expression
   command… ] …
[Else
   command… ]
End
```

- Process If . . . Else . . . End loop.

Lib Combine object files into a library file

```
Lib [option…] objectFile…  ≥ progress
```

- Combine the object files specified in the *objectFile* parameter into a library file.

-d	Suppress duplicate definition warnings.
-df *deleteFile*	Delete modules listed in file *deleteFile*.
-dm name[,*name*]…	Delete external modules and entry points.
-dn name[,*name*]…	Delete external names, making them local.

Lib (continued)

-mf	Use MultiFinder temporary memory if necessary.
-o *name*	Write object file *name* (default Lib.Out.o).
-p	Write progress information to diagnostics.
-sg *newSeg=old*[,*old*]...	Merge old segments into new segment.
-sn *oldSeg=newSeg*	Change segment name *oldSeg* to *newSeg*.
[Off]	Omit symbolic information; can be followed by:
-sym [On | Full]	Keep symbolic information (default).
[,NoLabels]	Discard label information.
[,NoLines]	Discard source line information.
[,NoTypes]	Discard type information.
[,NoVars]	Discard variable information.
-ver *n*	Set OMF file version number to *n*.
-w	Suppress warnings.

Line Find line in the target window

```
Line n
```

- Find line number *n* in the target window.

Link Link an application, tool, or resource

```
Link [option…] objectFile… > map ≥ progress
```

- Link *objectFile*.

-ac *n*	Align code modules to *n* byte boundaries.
-ad *n*	Align data modules to *n* byte boundaries.
-c *creator*	Set resource file creator to *creator* (default is '????').
-d	Suppress duplicate definition warnings.
-da	Desk accessory : Add NULL to segment names.
-f	Allow FORTRAN-style common data.
-l	Write a location map to output.
-la	List anonymous symbols in location map.

Link	(continued)	
	-lf	Include file and location of definitions in location map.
	-m *mainEntry*	Use *mainEntry* as main entry point.
	-ma *name=alias*	create an alias (*alias*) for module *name*.
	-map	Write a full location map.
	-mf	Use MultiFinder temporary memory if necessary.
	-msg keyword[,...]	Enable or suppress certain warning and error messages: [no]dup — Suppress duplicate-symbol warnings. [no]multiple — Suppress multiple label error messages. [no]warn — Suppress warning messages.
	-o *outputFile*	Place linker output in *outputFile* (default file is Link.Out).
	-opt [Off]	Disable Object Pascal optimizations (default). Also: [On] — Enable optimizations. [NoBypass] — Enable optimizations, but always dispatch. [,Info] — Always go through jump table. [,Names] — Include MacsBug symbols.
	-p	Write progress information to diagnostics.
	-ra [*seg*]=*attr*[,*attr*...]	Set resource attributes of segment *seg*. The *attr* parameter can be expressed as: $xx (or) *nnn* (a hexadecimal or decimal number) or as a named attribute: resSysHeap resPurgeable resLocked resProtected resPreload resChanged (essentially ignored in this usage)
	-rn	Don't include resource names in resourceFile.
	-rt *type=id*	Set resource type and lowest ID.
	-sg *newSeg*= *oldSeg*[,*oldSeg*]...	Merge old segments into new segment.

Link (continued)

-sn *oldSeg=newSeg*	Change segment name *oldSeg* to *newSeg*.
-srt	Sort global data by "near" and "far" references.
-ss *size*	Maximum segment size (default is 32760).
-sym [Off]	Disable symbolic output (default). Other parameters: [On I Full] Enable symbolic output, can be followed by: [,NoLabels] Omit label information. [,NoLines] Omit source line information. [,NoTypes] Omit type information. [,NoVars] Omit variable information.
-t *type*	Set resource file type (default 'APPL').
-uf *unRefFile*	Write list of unreferenced modules to *unRefFile*.
-w	Suppress warnings.
-x *crossRefFile*	Write cross-reference to *crossRefFile*.

Loop Repeat commands until Break

```
Loop
    command...
End
```

- Repeat *command* forever or until a Break command is encountered.

Make Build up-to-date version of a program

```
Make [option...] target... > commands ≥ progress
```

- Build up-to-date version of file *target*.

-d *name*[=*value*]	Define variable *name* as *value* (overrides makefile).
-e	Rebuild everything, regardless of dates.
-f *makefile*	Read dependencies from *makefile* (default file MakeFile).

Make (continued)

-p	Write progress information to diagnostics.
-r	Write roots of dependency graph to output.
-s	Write structure of target dependencies to output.
-t	Touch dates of targets and prerequisites.
-u	Identify targets in *makefile* not reached in build.
-v	Write verbose explanations to diagnostics.
-w	Suppress warnings.

MakeErrorFile Create error message text file

`MakeErrorFile [option...] [file...]  < file > listing ≥ progress`

- Create a special error message file to retrieve error messages associated with error numbers.

-l	Write listing to standard output.
-o *objName*	Write to file or directory *objName*.
-p	Write progress information to diagnostics.

Mark Assign a marker to a selection

`Mark [-y | -n] selection name [window]`

- Assign the marker *name* to *selection* in specified window (default is target window).

-y	Replace existing marker (avoids dialog).
-n	Don't replace existing marker (avoids dialog).

Markers List markers

`Markers [-q] [window]`

- List markers in *window* (default is target window).

-q	Don't quote the marker names.

MatchIt Semi-intelligent language-sensitive bracket matcher

`MatchIt [-a[sm] | -p[ascal] | -c] [-h] [-l] [-n] [-v] [window]`

- Find all left delimiters in a C, Pascal, or assembly language program, and then match them with their corresponding right delimiters.

-a[sm]	Target language is Assembler.
-p[ascal]	Target language is Pascal.
-c	Target language is C.
-h	Highlight all characters enclosed by match.
-l	Highlight entire lines containing match.
-n	Generate error message if no match.
-v	Display MatchIt's version number.

MergeBranch Merge a branch revision onto the trunk

`MergeBranch file...`

- Merge the branch revision of the HFS file *file* onto the trunk.

ModifyReadOnly Enable a read-only Projector file to be edited

`ModifyReadOnly f`

- Make the read-only file *f* a write-enabled file.

Mount Mount volumes

`Mount drive...`

- Mount *drive*.

MountProject Mount projects

`MountProject ([-s] [-pp] [-q] [-r]) | [p]`

- Mount Project *p*.

-s	Print names only, not commands.
-pp	List mounted projects using project paths.
-q	Don't quote names with special characters.
-r	List projects recursively.

Move — Move files and directories

`Move [-y | -n | -c] [-p]` *`name… target`* `≥` *`progress`*

- Move file or directory *name* to *target*.

-y	Overwrite target files (avoids dialog).
-n	Don't overwrite target files (avoids dialog).
-c	Cancel if conflict occurs (avoids dialog).
-p	Write progress information to diagnostics.

MoveWindow — Move window to x, y location

`MoveWindow [`*`h v`*`] [-i] [`*`w`*`]`

- Move window *w*'s upper left-hand corner to x, y screen coordinates h, v. Default window is target window.

-i	Ignore positioning errors.

NameRevisions — Define a symbolic name

`NameRevisions [-u` *`user`*`] [-project` *`project`*`] [-public | -b] [-r] [[-only] |` *`name`*

`[[-expand] [-s] | [-replace] [-dynamic] [`*`name…`*` | -a]]].`

- Create a symbolic *name* to represent a set of revisions (used with Projector).

-u *user*	Name of current user.
-project *project*	Name of project that contains the revisions.
-public	Create a public name.
-b	Print both public and private names.
-r	Recursively execute the NameRevisions command.
-only	Only print the names, not the associated revisions.
-expand	Evaluate names to revision level before printing.

NameRevisions (continued)

-s	Print a single name per line.
-replace	Completely overwrite the previous definition of name.
-dynamic	Evaluate names to revision level when using not defining.
-a	Include all the files in the project.

New Open a new window

`New [name...]`

- Open a new window, optionally named *name*.

Newer Compare modification dates of files

`Newer [-c] [-e] [-q] file... target > newer`

- Compare modification dates of files *file* and *target*.

-c	Compare creation *dates* of file and *target*.
-e	Report file names with same modification date as target.
-q	Don't quote file names with special characters.

NewFolder Create a new folder

`NewFolder n...`

- Create a new folder named *n*.

NewProject — Create a new project

```
NewProject -w | -close | ([-u user] [-cs comment |
-cf file] proj)
```

- Create a new project named *proj*.

-w	Open the New Project window.
-close	Close the New Project window.
-u *user*	Name of current user.
-cs *comment*	A short description of the project.
-cf *file*	A comment (see -cs option) is contained in *file*.

Open — Open file(s) in window(s)

```
Open [-n | -r] [-t] [name…]
```

- Open file *name* and display its contents as a window.

-n	Open new file (default name Untitled).
-r	Open file for read-only use.
-t	Open file as the target window.

OrphanFiles — Remove Projector information from a list of files

```
OrphanFiles file…
```

- Remove Projector information from *file*.

Parameters — Write parameters

```
Parameters [parameter…]  > parameters
```

- Write the parameters of the Parameters command to standard or specified output. This command is used primarily to check lists of parameters for debugging purposes.

Pascal Invoke Pascal compiler

`Pascal [option…] [file…] < file ≥ progress`

- Compile Pascal program.

-b	Generate A5 references for procedure addresses.
-c	Syntax check only; don't create object file.
-clean	Erase all symbol table resources.
-d *name*=(TRUE | FALSE)	Set compile time variable *name*.
-e file	Write errors to file.
-forward	Allow only explicit forward and external object declarations.
-h	Suppress error messages regarding unsafe handles.
-i *directory,...*	Search for includes in *directory, . . .*
-k *directory*	Create symbol table resource files in *directory*.
-m	Allow greater than 32K globals by using 32-bit references.
-mbg ch8	Include v2.0 compatible MacsBug symbols.
-mbg full	Include full (untruncated) symbols for MacsBug.
-mbg off	Don't include symbols for MacsBug.
-mbg *number*	Include MacsBug symbols truncated to length *number*.
-mc68020	Generate MC68020 code.
-mc68881	Generate MC68881 code for floating-point operations.
-n	Generate separate global data modules.
-noload	Don't use or create any symbol table resources.
-o *objName*	Generate code in file or directory *objName*.
-ov	Generate code to test for overflow.
-p	Write progress information to diagnostics.
-r	Don't generate range checking code.
-rebuild	Rebuild all symbol table resources.
-sym off	Include SADE object file information.

Pascal	(continued)	
	-sym on | full	Generate symbolic debugger object records.
	-t	Write compilation time to diagnostics.
	-u	Initialize all data to $7267 for debugging use.
	-w	Turn off peephole optimizer.
	-y *directory*	Create temporary files in *directory*.

PasMat Pascal programs formatter

`PasMat [`*`option...`*`] [`*`input`*` [`*`output`*`]]  < `*`input`*` > `*`output`*` ≥ `*`progress`*

- Reformat Pascal source code into a standard format, suitable for printed listings or compilation. Options to this command are explained in the *MPW 3.0 Pascal Reference* and later MPW Pascal documentation.

-a	Set a- to disable CASE label bunching.
-b	Set b+ to enable IF bunching.
-body	Set body+ to disable indenting procedure bodies.
-c	Set c+ to suppress Return before BEGIN.
-d	Set d+ to use {...} comment delimiters.
-e	Set e+ to capitalize identifiers.
-entab	Replace multiple blanks with tabs.
-f	Set f- to disable formatting.
-g	Set g+ to group assignment and call statements.
-h	Set h- to disable FOR, WHILE, WITH bunching.
-i *directory,...*	Search for includes in *directory,...*.
-in	Set in+ to process includes.
-k	Set k+ to indent statements between BEGIN and END.
-l	Set l+ to literally copy reserved words, identifiers.
-list *file*	Write listings to file.

PasMat (continued)

-n	Set n+ to group format parameters.
-o *width*	Set output line width (default 80).
-p	Write progress information to diagnostics.
-pattern =*old*=*new*=	Modify include names, changing *old* to *new*.
-q	Set q+ to specify no special ELSE IF formatting.
-r	Set r+ to make reserved words uppercase.
-rec	Set rec+ to indent field lists under defined ID.
-s *file*	Rename identifiers based on names listed in *file*.
-t *tab*	Set output tab setting (default 2).
-u	Rename identifiers to match first occurrence.
-v	Set v+ to put THEN on separate line.
-w	Set w+ to make identifiers uppercase.
-x	Set x+ to suppress space around operators.
-y	Set y+ to suppress space around :=.
-z	Set z+ to suppress space after commas.
-:	Set :+ to align colons in VAR declarations.
-@	Set @+ to force multiple CASE tags on separate lines.
-∂	Set #+ for "smart" grouping of assignments and calls.
-_	Set _+ to delete _ from identifiers.

PasRef Pascal cross-referencer

`PasRef [`*`option…`*`] [`*`file…`*`] < `*`file`*` > `*`crossReference`*` ≥ `*`progress`*

- Create cross-reference listing of a Pascal source file.

-a	Process includes and units each time encountered.
-c	Process includes and units only once.
-d	Process each file separately.
-i *directory,…*	Search for includes in directory,….
-l	Write identifiers in lowercase.

PasRef (continued)

-n	Don't process USES or includes.
-ni | -noi[ncludes]	Don't process include files.
-nl | -nol[istings]	Don't list the input.
-nolex	Don't write lexical information.
-nt | -not[otal]	Don't write total line count.
-nu | -nou[ses]	Don't process USES declarations.
-o	Source written using Object Pascal.
-p	Write progress information to diagnostics.
-s	Don't write include and USES file names.
-t	Cross-reference by total line number.
-u	Write identifiers in uppercase.
-w *width*	Set output line width (default 110).
-x *width*	Set maximum identifier width.

Paste Replace selection with Clipboard contents

`Paste [-c count] selection [window]`

- Paste *selection* into *window* (default is target window).

-c *count*	Execute the Paste command *count* times.

PerformReport Generate a performance report

`PerformReport [option…] > reportFile ≥ progress`

- Read a link map text file and a performance data text file, and generate a report that reports performance data for procedure names.

-a	List all procedures in segment order (Defaults produce only partial list, sorted by %).
-l *linkDataFile*	Read link map file (concatenated with ROM.list).
-m *measurementsFile*	Read performance measurements file (default is file Perform.Out).
-n *nn*	Show the top *nn* procedures (default is 50).
-p	Write progress information to diagnostics.

Position

Display current line position

`Position [-l | -c] [window...]`

- Display current position of insertion point in target or specified *window*.

-l	List only the line number.
-c	List only the character offsets.

Print

Print text file

`Print [option...] file... < file ≥ progress`

- Print *file*.

-b	Print a border around the text.
-b2	Alternate form of border.
-bm *n*[.*n*]	Bottom margin in inches (default is 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-number field; - causes padding with 0s.
-p	Write progress information to diagnostics.
-page *n*	Number pages beginning with *n*.
-ps *filename*	Include PostScript file as background for each page.
-q *quality*	Print quality (options: HIGH, STANDARD, DRAFT).

Print (continued)

-r	Print pages in reverse order.
-rm *n*[.*n*]	Right margin in inches (default is 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 is 0).
-to *n*	Stop printing after page *n*.

ProcNames Display Pascal procedure and function names

```
ProcNames [option…] [file…]  < file ≥ progress
```

- Display procedure and function names in a Pascal source file.

-c	Process includes and units only once.
-d	Reset total line count to 1 on each new file.
-e	Suppress page eject between each procedure listing.
-f	PasMat format compatibility mode.
-i *directory,…*	Search for includes or USES in *directory,…*.
-n	Suppress line number and level information.
-o	Source file is an Object Pascal program.
-p	Write progress information to diagnostics.
-u	Process USES declarations.

Project Set or write the current project

```
Project [-q | projectName] > project
```

- Set the current project to *projectName*. If *projectName* is omitted, list the current project.

-q	Don't quote project name if it includes special characters.

ProjectInfo Display information about a project

```
ProjectInfo [-project project] [-comments] [-latest]
[-f] [-r] [-s] [-only | -m]
[-af author | -a author] [-df dates | -d dates]
[-cf pattern | -c pattern]
[-t pattern] [-n name] [-newer | -update] [file…].
```

- List information about each revision in the current project's revision tree(s).

-a *author*	List only revisions created by *author*.
-af *author*	List only files created by *author*.
-c *pattern*	List only revisions whose comment contains *pattern*.
-cf *pattern*	List only files whose comment contains *pattern*.
-comments	List comments along with the rest of the information.
-d *dates*	List only revisions whose create date is within *dates*. Format of *dates* is *mm/dd/yy* [[*hh:mm*[:*ss*]] [AM | PM]]. The *dates* parameter may take these forms: *dates* On date specified by *dates*. < *dates* Before but not including *dates*. ≤ *dates* Before and including *dates*. > *dates* After and not including *dates*. ≥ *dates* After and including *dates*. *date-date* Between and including *dates*.
-df *dates*	List only files whose mod date is within *dates*.
-f	List file information.
-log	Print project log.
-m	List only files/revisions that are checked out.
-newer	List information on newest files in the current project.
-only	List only project information.
-project *proj*	Name of project to get information on.
-r	Recursively list subprojects.
-latest	List only information on the latest revision on the main trunk.

ProjectInfo (continued)

-s	Short listing, names and revision names only.
-t *pattern*	List only revisions whose task contains *pattern*. (Pattern is either a literal string or /regular expression/).
-n *name*	List only revisions that have *name*. The *name* parameter may take these forms:.
name	In name.
	< *name* — Before *name*.
	≤ *name* — Before and including *name*.
	> *name* — After *name*.
	≥ *name* — After and including *name*.
-update	List information on new files in current project directory.

Quit — Quit MPW

```
Quit [-y | -n | -c]
```

- Quit and exit MPW.

-y	Save all modified windows (avoids dialog).
-n	Do not save any modified windows (avoids dialog).
-c	Cancel if a window needs to be saved (avoids dialog).

Quote — Echo parameters, quoting if needed

```
Quote [-n] [parameter…]  > parameters
```

- Write *parameter*, enclosing parameters that contain special characters in quotes.

-n	Don't write return following the parameters.

Rename Rename files and directories

`Rename [-y | -n | -c] oldName newName`

- Rename file or directory *oldName*.

-y	Overwrite existing file (avoids dialog).
-n	Don't overwrite existing file (avoids dialog).
-c	Cancel if conflict occurs (avoids dialog).

Replace Replace the selection

`Replace [-c count] selection replacement [window]`

- Replace *selection* text with *replacement* text in target or selected *window*.

-c *count*	Execute the Replace command *count* times.

Request Request text from a dialog box

`Request [-q] [-d default] [message...] < file`

- Display a Request dialog window, and accept user input.

-q	Don't set status if user selects cancel.
-d *default*	Set default response.

ResEqual Compare the resources in two files

`ResEqual [-p] file1 file2`

- Compare *file1*'s resources with those of *file2*.

-p	Write progress information to diagnostics.

Revert Revert window to previously saved state

`Revert [-y] [window...]`

- Revert target or specified *window* to previously saved state.

-y	Revert to old version (without dialog).

Rez

Resource compiler

`Rez [option…] [fileName…] < file ≥ progress`

- Compile resource definition file *fileName*.

-a[ppend]	Merge resource into output resource file.
-align word | longword	Align resource to word or longword boundaries.
-c[reator] *creator*	Set output file creator.
-d[efine] *name*[=*value*]	Same as #define *value*.
-i[nclude] *pathname*	Path to search when looking for #include files.
-m[odification]	Don't change the output file's modification date.
-o *file*	Write output to *file* (default is Rez.Out).
-ov	OK to overwrite protected resources when appending.
-p	Write progress information to diagnostics.
-rd	Suppress warnings for redeclared types.
-ro	Set the mapReadOnly flag in output.
-s[earch] *pathname*	Path to search when looking for INCLUDE resources.
-t[ype] *type*	Set output file type.
-u[ndef] *name*	Same as #undef *name*.

RezDet

Detect inconsistencies in resources

`RezDet [option…] fileName… > dump`

- Find and report inconsistencies or errors in resource definition file *fileName*.

-b[ig]	Read resources one at a time, not all at once.
-d[ump]	Write -show information, plus headers, lists, etc.
-l[ist]	Write list of resources with minimum information.
-q[uiet]	Don't write any output, just set {Status}.
-r[awdump]	Write -dump information plus contents.
-s[how]	Write information about each resource.

RotateWindows Send active (frontmost) window to back

```
RotateWindows [-r]
```

- Rotate windows, sending active (frontmost) window to back. MPW 3.2 adds the capability of reverse rotation, bringing back window to front (-r option).

-r	Reverse rotation; bring back window to front (MPW 3.2)

Save Save specified windows

`Save [-a | ` *`window...`*`]`

- Save target window or specified *window.*

SaveOnClose Set window-saving preference (MPW 3.2)

`SaveOnClose [-a | -d | -n] [`*`window`*`]`

- Save, do not save, or ask whether to save when closing the target window or the specified *window.*

-a	Always save window when closing.
-d	Default (Display "Yes/No/Cancel" dialog).
-n	Never save window when closing.

Search Search files for pattern

`Search [-s | -i] [-r] [-q] [-f `*`file`*`] `*`pattern`*` [`*`file...`*`]`
`< `*`file`*` > `*`found`*

- Search for *pattern* in target window or specified *file.*

-b	Break "File/Line" from matched pattern (MPW 3.2).
-f *file*	Lines not written to output are put in this file.
-i	Case-insensitive search (overrides {CaseSensitive}).
-nf	Write "pattern not found" to standard error and set status = 2 (MPW 3.2).

Search (continued)

-q	Suppress file name and line number in output.
-r	Write nonmatching line to standard output.
-s	Case-sensitive search (overrides {CaseSensitive}).

Set Define or write shell variables

`Set [name [value]] > variableList`

- Assign the string *value* to the variable *name*. If *value* is omitted, write the variable *name* and its value to standard or specified output. If both *name* and *value* are omitted, write a list of all variables and their values to standard or specified output.

SetDirectory Set the default directory

`SetDirectory dir`

- Set default directory to *dir*.

SetFile Set attributes

`SetFile [option…] objectName…`

- Set attributes of file or directory *objectName*.

-a *attributes*	Set attributes (lowercase = 0, uppercase = 1).* These attributes may be used: L Locked. V Invisible.* B Bundle. S System. I Inited.* D Desktop.* M Shared (can run multiple times). A Always switch launch (if possible).
-c *creator*	File creator.

SetFile (continued)

-d *date*	Creation date (mm/dd/yy [hh:mm[:ss] [AM I PM]]).* (Period [.] represents the current date and time).
-l *h,v*	ICON location (horizontal,vertical).*
-m *date*	Modification date (mm/dd/yy [hh:mm[:ss] [AM I PM]]).*
-t *type*	File type.

* Allowed with folders.

SetPrivilege Set access privileges for directories on file servers

`SetPrivilege [option…] directory… > information`

- Set access privileges for *directory*.

-d *privileges*	Set privileges for seeing directories.
-f *privileges*	Set privileges for seeing files.
-g *group*	Make the directories belong to *group*.
-i	Return information on directories.
-m *privileges*	Set privileges for making changes.
-o *owner*	Make *owner* the owner of directories.
-r	Operate (set or list) recursively.

The following privilege characters may be used with the -d, -f, or -m options (uppercase enables the privilege, lowercase disables it):

O	Owner.
G	Group.
E	Everyone.

SetVersion

Maintain version and revision number

```
SetVersion [option…] file  > output ≥ progress
```

- Set the version of *file*. For full explanations of options, see the *MPW 3.0 Reference*.

-b	Increment the bug fix component by 1.
-country *name*	Country code name.
-csource *file*	Update the #define version string in C source file.
-d	Display (updated) version numbers to standard output.
-fmt n*f*.m*f*	Format version numbers according to specification.
-i *resID*	Use specified resource ID instead of 0.
-p	Write SetVersion's version information to diagnostic file.
-prefix *pfx*	Prefix version with specified *pfx*.
-[p]source *file*	Update the Version string constant in source file *file*.
-r	Increment the revision component by 1.
-rezsource *file*	Update the resource definition in Rez source file *file*.
-sb *bugfix*	Set the bug fix component to the specified value.
-sr *revision*	Set the revision component to the specified value.
-stage *stage*	Set release stage for a 'vers' resource.
-suffix *suffix*	Append *suffix* to version.
-sv *version*	Set the version component to the specified value.
-sx *nonrel*	Set the nonrelease component to the specified value.
-sync 1 | 2	Synchronize 'vers',1 with 'vers',2 or vice versa.
-t *type*	Use specified resource type.
-v	Increment the version component by 1.
-verid *identifier*	Use C/Pascal source version id instead of "version".
-version *fmtstring*	Alternate way of specifying version component actions.
-verstring *longstring*	Set the long version string of a Finder 'vers' resource.
-x	Increment the nonrelease component by 1.

Shift Renumber command file positional parameters

`Shift [`*number*`]`

- Increment command script positional parameters by *number*. For example, if *number* is 1, change parameters {1}, {2}, etc., to {1+1}, {2+1}, etc.

ShowSelection Show selection relative to window position (MPW 3.2)

`ShowSelection [-t | -b | -c | -n `*num*` | -l `*num*`]`
`[`*window*`]`

- Show selection at specified location in target window or specified *window*.

-t	Pin selection to top of window.
-b	Pin selection to bottom of window.
-c	Pin selection to center of window.
-n *num*	Move selection to *num* lines from top of window.
-l *num*	Move line number *num* to top of window.

Shutdown Power down or restart computer

`Shutdown [-y | -n | -c] [-r]`

- Shut down the computer. If -r option is used, restart.

-y	Save all modified windows (avoids dialog).
-n	Do not save any modified windows (avoids dialog).
-c	Cancel if a window needs to be saved (avoids dialog).
-r	Restart the machine.

SizeWindow Set a window's size

`SizeWindow[`*h v*`] [`*window*`]`

- Resize target window or specified *window* to *h* horizontal pixels by *v* vertical pixels.

Sort Sort or merge lines of text

```
Sort [option…] [files…]
```

- Sort or merge lines of text in the target window or in specified *files*, in accordance with options.

-b	Skip leading blanks of each field.
-check	Check if input is sorted (exit code 5 if not).
-d	Sort fields as decimal numbers.
-f *field*[,*field*]	Specify fields to sort on. The *field* parameters take these forms: [F][.C][-K][bdlqrtux] or [F][.C][+N][bdlqrtux] F Field number: 0 = whole line [default]. 1 = first word. 2 = second word… C Starting column number (from 1); default = 1. K Ending column number (> = C); default = infinite. N Maximum number of characters in the field; default = infinite. Only one of -K or +N can be specified.
-l	Convert to lowercase before comparison.
-merge	Merge presorted input files.
-o *file*	Specify output file (command allows sorting in place).
-p	Print version and progress information.
-quote	Handle fields with quotes.
-r	Reverse order of comparison.
-stdin	Placeholder for standard input (acts like a file).
-t	Sort fields as text (default).
-u	Convert to uppercase before comparison.
-unique	Write only unique output lines.
-x	Sort fields as hexadecimal numbers.

StackWindows Arrange windows diagonally with title bars showing

`StackWindows [-h num] [-v num] [-r t,l,b,r] [-i | windows…]`

- Stack screen windows in a diagonal, top-to-bottom pattern, with only the title bars of inactive windows showing.

-h *num*	Horizontal offset between windows.
-v *num*	Vertical offset between windows.
-r *t,l,b,r*	Rectangle (top, left, bottom, right) in which to stack windows.
-i	Include the worksheet.

Target Make a window the target window

`Target name`

- Make window *name* the target window.

TileWindows Arrange windows in a tiled fashion

`TileWindows [-h | -v] [-r t,l,b,r] [-i | windows…]`

- Tile *windows*.

-h	Tile windows horizontally.
-v	Tile windows vertically.
-r *t,l,b,r*	Rectangle (top, left, bottom, right) in which to tile windows.
-i	Include the worksheet window in the tiling operation.

TransferCkid Move Projector information from one file to another

`TransferCkid sourceFile destinationFile`

- Move Projector information from *sourceFile* to *destinationFile*.

Translate — Translate characters

`Translate [-p] [-s] src [dst] < file > output ≥ progress`

- Copy standard or selected input to standard or selected output, with characters specified in the parameter string *src* mapped into the parameter string *dst;* all other characters are copied as is. For more detailed information on this command, see the *MPW 3.0 Reference Manual.*

-p	Write progress information to diagnostics.
-s	Set font, font size, and tab setting of output.

Unalias — Remove aliases

`Unalias [name...]`

- Remove any alias definition associated with alias *name*. **Caution**: If *name* is not specified, all aliases are removed.

Undo — Undo the last edit

`Undo [window]`

- Undo the last edit in target window or specified *window:*

Unexport — Remove variable definitions from the export list

`Unexport [-r | -s | name...] > unexports`

- Remove the specified variables from the list of exported variables.

-r	Generate Export commands for all unexported variables.
-s	Print the names only.

Unmark — Remove a marker from a window

`Unmark markerName... windowName`

- Remove the *markerName* marker from the *windowName* window.

Unmount Unmount volumes

`Unmount` *`volume...`*

- Unmount *volume.*

UnmountProject Unmount projects

`UnmountProject -a |` *`projectName...`*

- Unmount Project *projectName.*

-a Unmount all mounted projects.

Unset Remove shell variable definitions

`Unset [`*`name...`*`]`

- Remove any variable definitions associated with *name.* **Caution:** If no *name* is specified, all variable definitions are removed.

UserVariables Use Commando to set user variables

`UserVariables`

- Display UserVariables Commando; Commando is used to set user variables.

Volumes List mounted volumes

`Volumes [-l] [-q] [`*`volumeName...`*`] >` *`volumeList`*

- List volume name and any other information requested for volume *volumeName.* If *volumeName* is not specified, all mounted volumes are listed.

-l Long format (name, drive, size, free, files, directories).

-q Don't quote volume names with special characters.

WhereIs Find the location of a file

```
WhereIs [-c] [-d] [-v] [-s directory]… pattern
```

- Find and report location of any file that contains *pattern* as part of its file name.

-c	Completely match file pattern.
-d	Include directories.
-v	Verbose output: Put summary line at end of listing.
-s *objectName*	Start search with directory or volume *objectName*.

Which Determine which file the shell will execute

```
Which [-a] [-p] [name] > file ≥ progress
```

- Determine what command the shell will execute when command or alias *name* is entered.

-a	Report all commands named *name*.
-p	Write progress information to diagnostics.

Windows List windows

```
Windows [-q]
```

- List windows.

-q	Don't quote window names with special characters.
-o	Write out the "Open…" commands (MPW 3.2).

ZoomWindow Enlarge or reduce a window's size

```
ZoomWindow [-b | -s] [windowName]
```

- Zoom window *windowName*.

-b	Zoom to full screen (full size).
-s	Zoom back to regular size (reduced size).

Appendix B
Commands Arranged by Category

Editing Commands

Adjust	Adjust lines
Clear	Clear the selection
Copy	Copy selection to Clipboard
Count	Count lines and characters
Cut	Copy selection to Clipboard and delete it
Entab	Convert runs of spaces to tabs
Find	Find and select a text pattern
Format	Set or display formatting options for a window
Line	Find line in the target window
Mark	Assign a marker to a selection
Markers	List markers
MatchIt	Semi-intelligent language-sensitive bracket matcher
Paste	Replace selection with Clipboard contents
Position	Display current line position
Replace	Replace the selection
Revert	Revert window to previously saved state
Search	Search files for pattern
Sort	Sort or merge lines of text
Translate	Translate characters
Undo	Undo the last edit
Unmark	Remove a marker from a window

File and Directory Commands

Backup	Folder file backup
Catenate	Concatenate files
Compare	Compare text files
CompareFiles	Compare text files and interactively view differences
CompareRevisions	Compare two revisions of a file in a project
Delete	Delete files and directories
Directory	Set or write the default directory
Duplicate	Duplicate files and directories
DuplicateIIGS	Copy files between Mac and GS/OS volumes
Equal	Compare files and directories
Exists	Confirm the existence of a file or directory
ExpressIIGS	Convert file(s) from OMF to ExpressLoad format
FileDiv	Divide a file into several smaller files
Files	List files and directories
Move	Move files and directories
Newer	Compare modification dates of files
NewFolder	Create a new folder
Open	Open file(s) in window(s)
Rename	Rename files and directories
Save	Save specified windows
SetDirectory	Set the default directory
SetFile	Set file/folder attributes
SetPrivilege	Set access privileges for directories on file servers
SetVersion	Maintain version and revision number
WhereIs	Find the location of a file

Macintosh/Apple IIGS Programming Commands

Asm	Assemble a program
AsmIIGS	Assemble an Apple IIGS program
AsmCvtIIGS	Convert APW Assembler source files to AsmIIGS format
AsmMatIIGS	Assembler source formatter
BuildCommands	Show build commands
BuildIndex	Create an index for a data file
BuildProgram	Build the specified program
C	Compile a C program
CIIGS	Compile MPW IIGS C program
Canon	Canonical spelling tool
CFront	C++ to C translator
CPlus	Script to compile C++ source
CreateMake	Create a simple makefile
CreateMakeIIGS	Create Make files that build IIGS programs
DeleteNames	Delete user-defined symbolic names
DumpCode	Write formatted resources
DumpFile	Display contents of any file
DumpObj	Write formatted object file
DumpObjIIGS	Dump OMF files
GetErrorText	Display error messages based on message number
Lib	Combine object files into a library file
Link	Link an application, tool, or resource
LinkIIGS	The MPW IIGS Linker
Make	Build up-to-date version of a program
MakeBinIIGS	Convert load files to binary files
MakeErrorFile	Create error message text file
MakeLibIIGS	Create IIGS Library files
Pascal	Compile Pascal program
PascalIIGS	The MPW IIGS Pascal Compiler
PasMat	Pascal programs formatter
PasRef	Pascal cross-referencer
PerformReport	Generate a performance report
ProcNames	Display Pascal procedure and function names

Menu Commands

AddMenu	Add a menu item
BuildMenu	Create the Build menu
BuildMenuIIGS	Add CreateMakeIIGS to the Build menu
DeleteMenu	Delete user-defined menus and menu items
DirectoryMenu	Create the Directory menu

Printing and Disk-Drive Commands

Choose	Choose or list network file server volumes and printers
Eject	Eject volumes
Erase	Initialize volumes
Mount	Mount volumes
Print	Print text file
Unmount	Unmount volumes
Volumes	List mounted volumes

Projector Commands

CheckIn	Check a file into a project
CheckOut	Check a file out from a project
CheckOutDir	Specify directory where checked-out files will be placed
DeleteRevisions	Delete previous revisions of files in a project
MergeBranch	Merge a branch revision onto the trunk
ModifyReadOnly	Enable a read-only Projector file to be edited
MountProject	Mount projects
NameRevisions	Define a symbolic name
NewProject	Create a new project
OrphanFiles	Remove Projector information from a list of files
Project	Set or write the current project
ProjectInfo	Display information about a project
TransferCkid	Move Projector information from one file to another
UnmountProject	Unmount projects

Resource Commands

DeRez	Resource decompiler
DeRezIIGS	Resource decompiler for Apple IIGS
ResEqual	Compare the resources in two files
ResEqualIIGS	Compare resources in two Apple IIGS files
Rez	Resource compiler
RezIIGS	Resource compiler for Apple IIGS
RezDet	Detect inconsistencies in resources

Shell Programming Commands

Alias	Define or write command aliases
Beep	Generate tones
Begin	Group commands
Break	Break from For or Loop
Browser	Display MPW Browser tool (MPW 3.2)
Continue	Continue with next iteration of For or Loop
Date	Write the date and time
DoIt	Highlight and execute a series of shell commands
Echo	Echo parameters
Evaluate	Evaluate an expression
Execute	Execute command file in the current scope
Exit	Exit from a command file
Export	Make variables available to commands
Flush	Flush tools that the shell has cached
For	Repeat commands once per parameter
Get	Get information about a keyword from a data file
Help	Write summary information
If	Conditional command execution
Loop	Repeat commands until Break
Parameters	Write parameters
Quit	Quit MPW
Quote	Echo parameters, quoting if needed

Shell Programming Commands (continued)

Set	Define or write shell variables
Shift	Renumber command file positional parameters
Shutdown	Power down or restart computer
Unalias	Remove aliases
Unexport	Remove variable definitions from the export list
Unset	Remove shell variable definitions
UserVariables	Use Commando to set user variables
Which	Determine which file the shell will execute

Window and Dialog Commands

Alert	Display an alert box
Align	Align text to left margin
Close	Close specified windows
Commando	Display a dialog interface for commands
Confirm	Display a confirmation dialog box
GetFileName	Display a Standard File dialog box
GetListItem	Display items for selection in a dialog box
MoveWindow	Move window (to horizontal, vertical location)
New	Open a new window
Request	Request text from a dialog box
RotateWindows	Send active (frontmost) window to back
SaveOnClose	Set window-saving preference (MPW 3.2)
ShowSelection	Show selection at specified place in window (MPW 3.2)
SizeWindow	Set a window's size
StackWindows	Arrange windows with title bars showing
Target	Make a window the target window
TileWindows	Arrange windows in a tiled fashion
Windows	List windows
ZoomWindow	Enlarge or reduce a window's size

Appendix C
The Creation.p Program

```
PROGRAM Creation;

    USES MemTypes, QuickDraw, OSIntf, ToolIntf,
    PackIntf, Traps, PrintTraps;

        {Functions and procedures}

    FUNCTION IsAppWindow(window: WindowPtr): BOOLEAN;
        FORWARD;

    FUNCTION GetSleep: LONGINT;
        FORWARD;

    PROCEDURE AboutDialog;
        FORWARD;

    PROCEDURE AdjustMenus;
        FORWARD;

    PROCEDURE DoActivate(becomingActive: BOOLEAN);
        FORWARD;

    PROCEDURE DoKey;
        FORWARD;
```

```
PROCEDURE DoMenu(result: LONGINT);
    FORWARD;

PROCEDURE DoUpdate;
    FORWARD;

PROCEDURE Initialize;
    FORWARD;

PROCEDURE PrintDoc;
    FORWARD;

PROCEDURE SetupMenus;
    FORWARD;

PROCEDURE UpdateActive;
    FORWARD;

PROCEDURE UpdateRects;
        FORWARD;

PROCEDURE FatalError(error: INTEGER);
    FORWARD;

PROCEDURE AlertUser(error: INTEGER);
    FORWARD;

FUNCTION TrapAvailable(tNumber: INTEGER; tType:
    TrapType): BOOLEAN;
    FORWARD;

PROCEDURE EventLoop;
    FORWARD;

PROCEDURE AdjustCursor;
    FORWARD;

PROCEDURE DoCloseWindow;
    FORWARD;

PROCEDURE DoOpenWindow;
    FORWARD;
```

```
CONST
    kSysEnvironsVersion = 1; {Tells SysEnvirons
        what kind of SysEnvRec we understand}
    kOSEvent = app4Evt; {event used by MultiFinder}
    kSuspendResumeMessage = 1; {high byte of
        suspend/resume event message}
    kResumeMask = 1;
    kMouseMovedMessage = $FA;
    kMinHeap = 29 * 1024;
    kMinSpace = 20 * 1024; {Minimum memory needed
        for app to run}
    kErrStrings = 128; {Resource ID for STR#
        resource}

    eWrongMachine = 1; {Indicies into STR# resources}
    eSmallSize = 2;
    eNoMemory = 3;
    eNoSpacePaste = 8;

    {*** Resources ***}

    rMenuBar = 128; {application's menu bar}
    rUserAlert = 129; {user error alert}

    {*** Menu constants ***}

    mApple = 128; {Apple menu}
    iAbout = 1;

    mFile = 129; {File menu}
    iNew = 1;
    iOpen = 2;
    iClose = 4;
    iPageSetup = 9;
    iPrint = 10;
    iQuit = 12;

    mEdit = 130; {Edit menu}
    iUndo = 1;
    iCut = 3;
    iCopy = 4;
    iPaste = 5;
    iClear = 6;
```

```
        iSelectAll = 8;

        mFont = 131; {Font menu (program fills in)}

        mSize = 132; {Size menu (program fills in)}

        mStyle = 133; {Style menu}
        iPlain = 1;
        iBold = 2;
        iItalic = 3;
        iUnderline = 4;
        iOutline = 5;
        iShadow = 6;

    VAR
        gStyle: TextStyle;
        gMenu: MenuHandle;
        gMac: SysEnvRec; {set up by Initialize}
        gHasWaitNextEvent: BOOLEAN; {set up by
            Initialize}
        gInBackground: BOOLEAN; {maintained by
            Initialize  and DoEvent}

        quit: BOOLEAN;
        shiftDown: BOOLEAN;
        theChar: Char;
        templ: LONGINT;

        mousePt: Point;
        dragRect: Rect;
        textRect: Rect;
        myEvent: EventRecord;
        myWindow: WindowPtr;
        theWindow: WindowPtr;

        iBeamHdl: CursHandle;

        textH: TEHandle;
        printH: THPrint;
        fontArray: ARRAY [1..64] OF INTEGER;
        sizeArray: ARRAY [1..32] OF INTEGER;
```

```
{ ********** EXECUTABLE CODE STARTS HERE ********** }

        {$S Main}

    PROCEDURE AboutDialog;

        VAR
            aRect: Rect;
            oldPort: GrafPtr;
            aWindow: WindowPtr;

        BEGIN
            GetPort(oldPort);
            WITH aRect DO
                BEGIN
                left := (screenbits.bounds.right -
                    screenbits.bounds.left) DIV 2 - 100;
                right := left + 200;
                top := (screenbits.bounds.bottom -
                    screenbits.bounds.top) DIV 2 - 50;
                bottom := top + 110;
                END;
            aWindow := NewWindow(NIL, aRect, '', TRUE,
                dBoxProc, Pointer( - 1), TRUE, 0);
            SetPort(aWindow);
            TextFont(systemFont);
            MoveTo(10, 40);
            DrawString('     Welcome to Creation!');
            MoveTo(24, 70);
            DrawString('By [Put your name here]');
            REPEAT
                SystemTask
            UNTIL Button;
            DisposeWindow(aWindow);
            SetPort(oldPort);
            FlushEvents(mUpMask + mDownMask, 0);
        END; {AboutDialog}

    {$S Main}

    PROCEDURE AdjustMenus;

        VAR
            flag: BOOLEAN;
            i: INTEGER;
```

```
            lineHeight: INTEGER;
            fontAscent: INTEGER;
            n: LONGINT;
            curStyle: TextStyle;
            name: Str255;
            item: StyleItem;
            mode: INTEGER; { current style }

        BEGIN

            {clear check marks from the text menus}
            gMenu := GetMHandle(mFont);
            FOR i := 1 TO CountMItems(gMenu) DO
                CheckItem(gMenu, i, FALSE);
            gMenu := GetMHandle(mSize);
            FOR i := 1 TO CountMItems(gMenu) DO
                CheckItem(gMenu, i, FALSE);
            gMenu := GetMHandle(mStyle);
                FOR i := 1 TO CountMItems(gMenu) DO
            CheckItem(gMenu, i, FALSE);

            gMenu := GetMHandle(mFont);
            FOR i := 1 TO CountMItems(gMenu) DO
                BEGIN
                gMenu := GetMHandle(mFont);
                IF fontArray[i] = gStyle.tsFont THEN
                    CheckItem(gMenu, i, TRUE);
                END;

            gMenu := GetMHandle(mSize);
            FOR i := 1 TO CountMItems(gMenu) DO
                BEGIN
                gMenu := GetMHandle(mSize);
                IF sizeArray[i] = gStyle.tsSize THEN
                    CheckItem(gMenu, i, TRUE);
                END;

            gMenu := GetMHandle(mStyle);
            mode := doFace;

            IF TEContinuousStyle(mode, gStyle, textH)
    THEN
                BEGIN
```

```
            CheckItem(gMenu, iPlain, gStyle.tsface =
                []);
            CheckItem(gMenu, iBold, bold IN
                gStyle.tsface);
            CheckItem(gMenu, iItalic, italic IN
                gStyle.tsface);
            CheckItem(gMenu, iUnderline, underline IN
                gStyle.tsface);
            CheckItem(gMenu, iOutline, outline IN
                gStyle.tsface);
            CheckItem(gMenu, iShadow, shadow IN
                gStyle.tsface);
            END
        ELSE
            BEGIN
            CheckItem(gMenu, iPlain, FALSE);
            CheckItem(gMenu, iBold, FALSE);
            CheckItem(gMenu, iItalic, FALSE);
            CheckItem(gMenu, iUnderline, FALSE);
            CheckItem(gMenu, iOutline, FALSE);
            CheckItem(gMenu, iShadow, FALSE);
            END; { IF }

    END; {AdjustMenus}

{$S Main}

PROCEDURE DoActivate(becomingActive: BOOLEAN);

    BEGIN
        IF WindowPtr(myEvent.message) = myWindow
              THEN
            BEGIN
            IF becomingActive THEN
                BEGIN
                TEActivate(textH);
                gMenu := GetMHandle(mEdit);

                DisableItem(gMenu, 1);
                END
            ELSE
                BEGIN
                TEDeactivate(textH);
```

```
                    gMenu := GetMHandle(mEdit);
                    EnableItem(gMenu, 1);
                    END;
                END;
        END; {DoActivate}

    {$S Main}

    PROCEDURE DoKey;

        BEGIN
            IF myWindow = FrontWindow THEN
                theChar := CHR(BAND(myEvent.message,
                    charCodeMask));
            TEKey(theChar, textH);

        END; {DoKey}

    {$S Main}

    PROCEDURE DoMenu(result: LONGINT);

        CONST
            doToggle = 32; {requires system 6.0}

        VAR
            bool: BOOLEAN;
            theItem: INTEGER;
            theMenu: INTEGER;
            temp: INTEGER;
            name: Str255;
            ht, ascnt: INTEGER;
            hack: INTEGER;

        BEGIN
            theItem := LoWord(result);
            theMenu := HiWord(result);
            InitCursor;
            CASE theMenu OF
                mApple: {Apple menu}
                    IF (theItem = 1) THEN
                        AboutDialog
                    ELSE
```

```
        BEGIN
        gMenu := GetMHandle(mApple);
        GetItem(gMenu, theItem, name);
        temp := OpenDeskAcc(name);
        SetPort(myWindow);
        END;
mFile: {File menu}
    CASE theItem OF
        iOpen: DoOpenWindow;
        iClose: DoCloseWindow;
        iPageSetup: bool :=
            PrStlDialog(printH);
        iPrint: IF PrJobDialog(printH)
            THEN PrintDoc;
        iQuit: quit := TRUE;
    END;
mEdit: {Edit menu}
    BEGIN
    IF NOT SystemEdit(theItem - 1) THEN
        CASE theItem OF
            iCut:
                  BEGIN {Cut}
                  templ := ZeroScrap;
                  TECut(textH);
                  END;
            iCopy:
                  BEGIN {Copy}
                  templ := ZeroScrap;
                  TECopy(textH);
                  END;
            iPaste: TEStylPaste(textH);
                  {Paste}
            iClear: TEDelete(textH);
                  {Clear}
            iSelectAll: TESetSelect(0,
                  32767, textH);
        END;
    END;
mFont: {Font menu}
    BEGIN
    gMenu := GetMHandle(mFont);
    GetItem(gMenu, theItem, name);
    GetFNum(name, temp);
```

```
                gStyle.tsFont := temp;
                TESetStyle(doFont, gStyle, TRUE,
                    textH);
                END;
            mSize: {Size menu}
                BEGIN
                gMenu := GetMHandle(mSize);
                GetItem(gMenu, theItem, name);
                StringToNum(name, templ);
                gStyle.tsSize := templ;
                TESetStyle(doSize, gStyle, TRUE,
                    textH);
                END;
            mStyle: {Style menu}
                BEGIN
                HiliteMenu(6);
                IF theItem = 1 THEN
                    BEGIN
                    gStyle.tsface := [];
                    TESetStyle(doFace, gStyle, TRUE,
                        textH);
                    END
                ELSE
                    BEGIN
                    gStyle.tsface := [];
                    BitSet(@gStyle.tsface, 9 -
                        theItem);
                    TESetStyle(doFace + doToggle,
                        gStyle, TRUE, textH);
                    END;
                END;
        END;
        HiliteMenu(0);
    END; {DoMenu}

{$S Main}

PROCEDURE DoCloseWindow;

    BEGIN
        HideWindow(myWindow);
        gMenu := GetMHandle(mFile);
        DisableItem(gMenu, iClose);
```

```
        EnableItem(gMenu, iOpen);
    END; {DoCloseWindow}

{$S Main}

PROCEDURE DoOpenWindow;

    BEGIN
        ShowWindow(myWindow);
        gMenu := GetMHandle(mFile);
        DisableItem(gMenu, iOpen);
        EnableItem(gMenu, iClose);
    END; {DoOpenWindow}

{$S Main}

PROCEDURE DoMouse;

    VAR
        thePart: INTEGER;

    BEGIN
        thePart := FindWindow(myEvent.where,
            theWindow);
        CASE thePart OF
            inMenuBar:
            BEGIN
            AdjustMenus;
            DoMenu(MenuSelect(myEvent.where));
            END;
        inSysWindow: SystemClick(myEvent, theWindow);
        inContent:
            BEGIN
            IF theWindow <> FrontWindow THEN
                SelectWindow(theWindow)
            ELSE IF theWindow = myWindow THEN
                BEGIN
                GlobalToLocal(myEvent.where);
                shiftDown := BAND(myEvent.modifiers,
                    shiftKey) <> 0;
                TEClick(myEvent.where, shiftDown,
                    textH);
                END;
```

```
                END;
            inDrag: DragWindow(theWindow, myEvent.where,
                dragRect);
            inGrow:
                BEGIN
                templ := GrowWindow(theWindow,
                    myEvent.where, screenbits.bounds);
                InvalRect(theWindow^.portRect);
                SizeWindow(theWindow, LoWord(templ),
                    HiWord(templ), FALSE);
                UpdateActive;
                END;
            inGoAway: IF TrackGoAway(theWindow,
                myEvent.where) THEN DoCloseWindow;
        inZoomIn, inZoomOut:
            IF TrackBox(theWindow, myEvent.where,
                thePart) THEN
                BEGIN
                ZoomWindow(theWindow, thePart, FALSE);
                UpdateActive;
                END;
            END;
        END; {DoMouse}

    {$S Main}

    PROCEDURE AdjustCursor; {give time to DAs, set
    cursor, flash cursor}

        BEGIN
            IF (myWindow = FrontWindow) THEN
                BEGIN
                GetMouse(mousePt);
                IF PtInRect(mousePt, textRect) THEN
                    SetCursor(iBeamHdl^^)
                ELSE
                    SetCursor(arrow);
                TEIdle(textH);
                END;
        END; {AdjustCursor}

    {$S Main}
```

```
PROCEDURE DoUpdate;

    BEGIN
        theWindow := WindowPtr(myEvent.message);
        IF theWindow = myWindow THEN
            BEGIN
            SetPort(theWindow);
            BeginUpdate(theWindow);
            EraseRect(theWindow^.portRect);
            TEUpdate(theWindow^.portRect, textH);
                {draw the text}
            DrawGrowIcon(theWindow);
            EndUpdate(theWindow);
            END;
    END; {DoUpdate}

{$S Initialize}

FUNCTION TrapAvailable(tNumber: INTEGER; tType:
TrapType): BOOLEAN;

    BEGIN
        IF (tType = ToolTrap) & (gMac.machineType >
            envMachUnknown) &
            (gMac.machineType < envMacII) THEN
        BEGIN {512KE, Plus, or SE}
        tNumber := BAND(tNumber, $03FF);
        IF tNumber > $01FF THEN {which means the tool
        traps}
        tNumber := _Unimplemented; {only go to $01FF}
        END;
    TrapAvailable := NGetTrapAddress(tNumber, tType)
        <> GetTrapAddress(_Unimplemented);
    END; {TrapAvailable}

PROCEDURE Initialize;

    VAR
        count, ignoreError: INTEGER;
        menuBar: Handle;
        total, contig: LONGINT;
        ignoreResult: BOOLEAN;
        event: EventRecord;
```

```
BEGIN

    gInBackground := FALSE;

    FlushEvents(everyEvent, 0);
    InitGraf(@thePort);
    InitFonts;
    InitWindows;
    InitMenus;
    TEInit;
    InitDialogs(NIL);
    InitCursor;
    PrOpen;
    printH := THPrint(newHandle(SizeOf(TPrint)));
    IF printH = NIL THEN DebugStr('Not enough
        memory for print record.');
    PrintDefault(printH);

    FOR count := 1 TO 3 DO {allow alert default
        button to be outlined}
        ignoreResult := EventAvail(everyEvent,
            event);

    ignoreError := SysEnvirons
        (kSysEnvironsVersion, gMac);

    {If the machine doesn't have at least 128K
        ROMs, exit.}
    IF gMac.machineType < 0 THEN FatalError
        (eWrongMachine);

    gHasWaitNextEvent := TrapAvailable
        (_WaitNextEvent, ToolTrap);

    IF ORD(GetApplLimit) - ORD(ApplicZone) <
        kMinHeap THEN
        FatalError(eSmallSize);

    {* ZeroScrap; *} {*** You can uncomment
        this--TEMPORARILY--for debugging***}
```

```
        PurgeSpace(total, contig);
        IF total < kMinSpace THEN
            IF UnloadScrap <> noErr THEN
                FatalError(eNoMemory)
            ELSE
                BEGIN
                PurgeSpace(total, contig);
                IF total < kMinSpace THEN FatalError
                (eNoMemory);
                END;

        {***** Now we set up our application's
        environment ******}

        SetupMenus;
        SetRect(dragRect, - 32767, - 32767, 32767,
            32767);
        {WITH screenBits.bounds DO SetRect
            (textRect,4,24,right-4,bottom-4);}
        WITH screenbits.bounds DO SetRect(textRect, 2,
            24, right - 2, bottom - 2);
        {InsetRect(textRect,5,20);}
        InsetRect(textRect, 5, 15);
        myWindow := NewWindow(NIL, textRect,
            'Creation', TRUE, zoomDocProc,
                Pointer( - 1), TRUE, 0);
        SetPort(myWindow);

        UpdateRects;
        TextFont(times);
        TextSize(18);
        textH := TEStylNew(textRect, textRect);
        TEAutoView(TRUE, textH);
        iBeamHdl := GetCursor(iBeamCursor);

        quit := FALSE;
    END; {Initialize}

{$S Main}

PROCEDURE PrintDoc; {print 1 page of text with its
styles}
```

```
        VAR
            aRect: Rect;
            printTE: TEHandle;
            printPort: TPPrPort;
            status: TPrStatus;

        BEGIN
            aRect := printH^^.rPaper;
            InsetRect(aRect, 72, 72);

            printPort := PrOpenDoc(printH, NIL, NIL);
            printTE := TEStylNew(aRect, aRect);
            IF printTE = NIL THEN DebugStr('Not enough
                memory to print TERec.');
            printTE^^.inPort := GrafPtr(printPort);

            {copy and paste our text and styles for print
                mgr}
            TESetSelect(0, 32767, textH);
            TECopy(textH);
            TESetSelect(0, 0, textH);
            TESetSelect(0, 0, printTE);
            TEStylPaste(printTE);

            PrOpenPage(printPort, NIL);
            TEUpdate(aRect, printTE); {draw text on the
                printer}
            PrClosePage(printPort);
            PrCloseDoc(printPort);
            TEDispose(printTE);
            IF printH^^.prJob.bJDocLoop = bSpoolLoop THEN
                PrPicFile(printH, NIL, NIL, NIL, status);
        END; {PrintDoc}

    {$S Main}

    PROCEDURE SetupMenus;

        VAR
            i, n: INTEGER;
            l: LONGINT;
            s: Str255;
```

```
        menuBar: Handle;

    BEGIN
        menuBar := GetNewMBar(rMenuBar); {read menus
            into menu bar}
        IF menuBar = NIL THEN
          FatalError(eNoMemory);
        SetMenuBar(menuBar); {install menus}
        DisposHandle(menuBar);
        AddResMenu(GetMHandle(mApple), 'DRVR'); {add
            DA names to Apple menu}
        DrawMenuBar;

        gMenu := GetMHandle(mFont);
        AddResMenu(gMenu, 'FONT');
        FOR i := 1 TO CountMItems(gMenu) DO
            BEGIN
            gMenu := GetMHandle(mFont);
            GetItem(gMenu, i, s);
            GetFNum(s, n);
            fontArray[i] := n;
            END;
        gMenu := GetMHandle(mSize);
        FOR i := 1 TO CountMItems(gMenu) DO
            BEGIN
            gMenu := GetMHandle(mSize);
            GetItem(gMenu, i, s);
            StringToNum(s, l);
            sizeArray[i] := l;
            END;

        gMenu := GetMHandle(mFont);
        FOR i := 1 TO CountMItems(gMenu) DO
            BEGIN
            gMenu := GetMHandle(mFont);
            GetItem(gMenu, i, s);
            GetFNum(s, n);
            fontArray[i] := n;
            END;
        gMenu := GetMHandle(mSize);
        FOR i := 1 TO CountMItems(gMenu) DO
            BEGIN
            gMenu := GetMHandle(mSize);
```

```
            GetItem(gMenu, i, s);
            StringToNum(s, l);
            sizeArray[i] := l;
            END;
    END; {SetupMenus}

{$S Main}

PROCEDURE UpdateActive;

    BEGIN
        InvalRect(myWindow^.portRect);
        UpdateRects;
        WITH textH^^ DO
            BEGIN
            destRect := textRect;
            viewRect := textRect;
            END;
        TECalText(textH);
    END; {UpdateActive}

{$S Main}

PROCEDURE UpdateRects;

    BEGIN
        textRect := thePort^.portRect;
        WITH textRect DO
            BEGIN
            left := left + 4;
            right := right - 20;
            bottom := bottom - 20;
            END;
    END; {UpdateRects}

{$S Main}

PROCEDURE FatalError(error: INTEGER);

    BEGIN
        AlertUser(error);
        ExitToShell;
    END; {FatalError}
```

```
{$S Main}

PROCEDURE AlertUser(error: INTEGER);
{ Display an alert dialog when an error occurs }

    VAR
        itemHit: INTEGER;
        message: Str255;

    BEGIN
        SetCursor(arrow);
        GetIndString(message, kErrStrings, error);
        ParamText(message, '', '', '');
        itemHit := Alert(rUserAlert, NIL);
    END; {AlertUser}

{$S Main}

PROCEDURE EventLoop;

    VAR
        cursorRgn: RgnHandle;
        gotEvent: BOOLEAN;
        ignoreResult: BOOLEAN;
        mouse: Point;
        key: Char;

    BEGIN
        cursorRgn := NewRgn; {we'll pass an empty
                     region to WNE the first time
                     thru}
        REPEAT
            IF gHasWaitNextEvent THEN
                ignoreResult := WaitNextEvent
                              (everyEvent, myEvent,
                              GetSleep,cursorRgn)
            ELSE
                BEGIN
                SystemTask;
                gotEvent := GetNextEvent(everyEvent,
                            myEvent);
                END;
```

```
AdjustCursor;
CASE myEvent.what OF
    mouseDown: DoMouse;
    keyDown, autoKey:
        BEGIN
        key := CHR(BAND
            (myEvent.message,
               charCodeMask));
        IF BAND(myEvent.modifiers, cmdKey)
        <> 0 THEN
            BEGIN { Command key down }
            IF myEvent.what = keyDown
                  THEN
                  BEGIN
                  AdjustMenus;
                  DoMenu(MenuKey(key));
                  END; {IF}
            END
        ELSE
            DoKey;
        END; {keyDown}

    activateEvt: DoActivate
    (BAND(myEvent.modifiers, activeFlag)
    <> 0);
    updateEvt: DoUpdate;
    nullEvent: IF (textH <> NIL THEN
        IF (FrontWindow = myWindow) THEN
            TEIdle (textH);
               kOSEvent:
        CASE BAND(BROTL(myEvent.message,
        8), $FF) OF
            kMouseMovedMessage:
            TEIdle(textH);
            kSuspendResumeMessage:
                  BEGIN
                  gInBackground := BAND
                  (myEvent.message,
                  kResumeMask) = 0;
                  DoActivate(NOT
                  gInBackground);
                  END;
        END;
```

```
            END;
        UNTIL quit;
        PrClose;
    END; {EventLoop}

{$S Main}

FUNCTION GetSleep: LONGINT;

    VAR
        sleep: LONGINT;
        window: WindowPtr;

    BEGIN
        sleep := MAXLONGINT; {default value for sleep}
        IF NOT gInBackground THEN
            BEGIN {if we are in front...}
            window := FrontWindow; {and the front
                window is ours...}
            IF IsAppWindow(window) THEN
                BEGIN
                WITH textH^^ DO
                    IF selStart = selEnd THEN {and
                    the selection is an insertion
                    point...}
                        sleep := GetCaretTime; {we
                        need to blink the insertion
                        point}
                END;
        END;
        GetSleep := sleep;
    END; {GetSleep}

{$S Main}

FUNCTION IsAppWindow(window: WindowPtr): BOOLEAN;

    BEGIN
        IF window = NIL THEN
            IsAppWindow := FALSE
        ELSE {application windows have windowKinds
```
=

```
    userKind (8)}
            WITH WindowPeek(window)^ DO IsAppWindow
:=
            (windowKind = userKind);
    END; {IsAppWindow}

{********** THIS IS THE MAIN SEGMENT **********}

PROCEDURE _DataInit;
    EXTERNAL;

{This routine is automatically linked in by the MPW
Linker. This external reference to it is done so that
we can unload its segment, %A5Init.}

{$S Main}

BEGIN
    UnloadSeg(@_DataInit); {note that _DataInit
    must not be in Main!}

    MaxApplZone; {expand the heap so code segments
    load at the top}

    Initialize; {initialize the program}
    UnloadSeg(@Initialize); {note that Initialize
    must not be in Main!}

    gStyle.tsFont := times;
    gStyle.tsface := [];
    gStyle.tsSize := 12;
    TESetStyle(doAll, gStyle, FALSE, textH);
    AdjustMenus;

    EventLoop; {call the main event loop}
END.
```

Appendix D
Creation.r Resource Description File for Creation.p

```
#include "Types.r";
#include "SysTypes.r"
#include "Creation.h"

resource 'MBAR' (rMenuBar, preload) {
    { mApple, mFile, mEdit, mFont, mSize, mStyle };
};

resource 'MENU' (mApple, preload) {
    mApple, textMenuProc,
    0b1111111111111111111111111111101,  /* disable
        dashed line, */enabled, apple,/* enable About
        and DAs*/
    {
        "About Creation",
            noicon, nokey, nomark, plain;
        "-",
            noicon, nokey, nomark, plain
    }
};
```

```
resource 'MENU' (mFile, preload) {
    mFile, textMenuProc,
    0b11111111111111111111101100001000,
    enabled, "File",
    {
        "New",
            noicon, "N", nomark, plain;
        "Open",
            noicon, "O", nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Close",
            noicon, "W", nomark, plain;
        "Save",
            noicon, "S", nomark, plain;
        "Save As…",
            noicon, nokey, nomark, plain;
        "Revert",
            noicon, nokey, nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Page Setup…",
            noicon, nokey, nomark, plain;
        "Print…",
            noicon, nokey, nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Quit",
            noicon, "Q", nomark, plain
    }
};

resource 'MENU' (mEdit, preload) {
    mEdit, textMenuProc,
    0b11111111111111111111111110111101,
    enabled, "Edit",
     {
        "Undo",
            noicon, "Z", nomark, plain;
        "-",
            noicon, nokey, nomark, plain;
        "Cut",
            noicon, "X", nomark, plain;
```

```
            "Copy",
                noicon, "C", nomark, plain;
            "Paste",
                noicon, "V", nomark, plain;
            "Clear",
                noicon, "B", nomark, plain;
            "-",
                noicon, nokey, nomark, plain;
            "Select All",
                noicon, "A", nomark, plain
        }
};

resource 'MENU' (mFont, preload) {
    mFont, textMenuProc,
    0b1111111111111111111111111111111,
    enabled, "Font",
    {}
};

resource 'MENU' (mSize, preload) {
    mSize, textMenuProc,
    0b1111111111111111111111111111111,
    enabled, "Size",
     {
            "6",
                noicon, nokey, nomark, plain;
            "9",
                noicon, nokey, nomark, plain;
            "10",
                noicon, nokey, nomark, plain;
            "12",
                noicon, nokey, nomark, plain;
            "14",
                noicon, nokey, nomark, plain;
            "18",
                noicon, nokey, nomark, plain;
            "24",
                noicon, nokey, nomark, plain;
            "36",
                noicon, nokey, nomark, plain;
            "48",
```

```
                noicon, nokey, nomark, plain;
            "60",
                noicon, nokey, nomark, plain;
            "72",
                noicon, nokey, nomark, plain
        }
};

resource 'MENU' (mStyle, preload) {
    mStyle, textMenuProc,
    0b1111111111111111111111111111111,
    enabled, "Style",
     {
        "Plain",
            noicon, nokey, nomark, plain;
        "Bold",
            noicon, nokey, nomark, plain;
        "Italic",
            noicon, nokey, nomark, plain;
        "Underline",
            noicon, nokey, nomark, plain;
        "Outline",
            noicon, nokey, nomark, plain;
        "Shadow",
            noicon, nokey, nomark, plain
    }
};

resource 'SIZE' (-1) {       /* MultiFinder-aware
application */
    dontSaveScreen,
    acceptSuspendResumeEvents,
    enableOptionSwitch,
    canBackground,
    multiFinderAware,
    backgroundAndForeground,
    dontGetFrontClicks,
    ignoreChildDiedEvents,
    not32BitCompatible,
    reserved, reserved, reserved, reserved, reserved,
        reserved, reserved,
    96*1024,
    64*1024
};
```

Appendix E
Creation.make Makefile for Creation.p

```
#    File:        Creation.make
#    Target:      Creation
#    Sources:     Creation.p Creation.r
#    Created:     Wednesday, June 6, 1990 8:38:51 AM


OBJECTS = Creation.p.o


Creation ƒƒ Creation.make Creation.r
    Rez Creation.r -append -o Creation

Creation ƒƒ Creation.make {OBJECTS}
    Link -w -t APPL -c '????' -sym on -mf ∂
        {OBJECTS} ∂
        "{Libraries}"Runtime.o ∂
        "{Libraries}"Interface.o ∂
        "{PLibraries}"SANELib.o ∂
        "{PLibraries}"PasLib.o ∂
        -o Creation
Creation.p.o ƒ Creation.make Creation.p
    Pascal -sym on Creation.p
```

▸ Afterword

Once upon a time, Apple had to apologize for the lack of development tools available for the Macintosh. It may be hard for you young whippersnappers to believe, but when Apple shipped the Macintosh back in 1948 . . . er, 1984, the only way to develop full-potency Macintosh software was to use assembly language, or buy a Lisa computer for many thousands of dollars and wire it to the Macintosh. Ah, the bad old days.

Now, Apple may have to apologize for certain features of its development systems (or not), but there is certainly no shortage of ways to create software for Macintosh computers. To be sure, Apple hardly has a monopoly on great Macintosh development systems. Many top commercial developers use third-party development tools with great success, and even some folks inside Apple are known to dip into the outside toolbox now and then.

(That's all fine, by the way, because Apple's business is not to make heaps of money selling development systems. It's to be sure that there are lots of great development systems to keep everyone happy.)

Meanwhile, back home on the ranch, Apple's Macintosh Programmer's Workshop is a fascinating piece of software. It's so broad, so deep, and so powerful that thousands of programmers use it productively every day, and yet may not know what the heck they're doing a lot of the time. That's OK, because MPW lets you be a user at many different levels. You might be a programmer on a team, using scripts and commands written by someone else to help you do your job. Or, you might be that beloved guru who makes the cool scripts that everybody else desires. Both are reasonable and real jobs.

In this book, Mark Andrews has introduced you to that wonderful world of MPW. He's shown you around the basic parts of MPW that you'll need to know in order to get going with real work. You've seen some friendly parts and some less-friendly (but incredibly flexible and powerful) parts along the way, and now you should be ready to start creating your own Macintosh monuments to great programming.

Sometime, while you're enjoying the power and flexibility that MPW provides, remember us lonesome pioneers from way back when and the hardships that we suffered. Why, when I was your age, I had to walk 14 miles in my bare feet in the snow just to link a desk accessory. . .

Scott Knaster
Macintosh Inside Out Series Editor

Bibliography

Allen, Daniel K. *On Macintosh Programming: Advanced Techniques.* Reading, MA: Addison-Wesley, 1990.

Guide to the Macintosh Family Hardware, Second Edition. Reading, MA: Addison-Wesley, 1990.

Human Interface Guidelines: The Apple Desktop Interface. Reading, MA: Addison-Wesley, 1987.

Hansen, Augie. *C Programming, a Complete Guide to Mastering the C Language.* Reading, MA: Addison-Wesley, 1990.

Inside Macintosh, Volumes I-VI. Reading, MA: Addison-Wesley, 1985-1991.

Inside Macintosh X-Ref. Reading, MA: Addison-Wesley, 1988.

Introduction to MacApp 2.0 and Object-Oriented Programming. Cupertino, CA: Apple Computer, Inc., 1990.

Kerninghan, Brian W., and Ritchie, Dennis M. *The C Programming Language,* Second Edition. Englewood Cliffs, NJ: Prentice-Hall, 1988, 1978.

Knaster, Scott. *How to Write Macintosh Software.* Indianapolis: Hayden Books, 1986.

MacApp 2.0 General Reference. Cupertino, CA: Apple Computer, Inc., 1989.

MacApp 2.0 Tutorial. Cupertino, CA: Apple Computer, Inc., 1989.

Macintosh Programmer's Workshop 3.0 Reference. Cupertino, CA: Apple Computer, Inc., 1985-1988.

Macintosh Programmer's Workshop 3.0 Assembler Reference. Cupertino, CA: Apple Computer, Inc., 1985–1988.

Macintosh Programmer's Workshop C 2.0 Reference. Cupertino, CA: Apple Computer, Inc., 1987.

Macintosh Programmer's Workshop Object Pascal, Version 3.1. Cupertino, CA: Apple Computer, Inc., 1989.

MPW C++. Cupertino, CA: Apple Computer, Inc., 1989.

Pritchard, Paul. *An Introduction to Programming Using Macintosh Pascal.* Reading, MA: Addison-Wesley, 1988.

Niguidula, David, and Van Dam, Andries. *Pascal on the Macintosh: A Graphical Approach.* Reading, MA: Addison-Wesley, 1987.

Programmer's Introduction to the Macintosh Family. Reading, MA: Addison-Wesley, 1988.

Savitch, Walter. *Pascal: An Introduction to the Art and Science of Programming,* Second Edition. Reading, MA: Addison-Wesley, 1987.

Schildt, Herbert. *C: The Complete Reference.* Berkeley: Osborne McGraw-Hill, 1987.

Tondo, Clovis L., and Gimpel, Scott E. *The C Answer Book.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

West, Joel. *Programming with the Macintosh Programmer's Workshop.* New York: Bantam Books, 1987.

Weston, Dan. *Elements of C++ Macintosh Programming.* Reading, MA: Addison-Wesley, 1990.

Williams, Steve. *Programming the 68030.* Reading, MA: Addison-Wesley, 1989.

Williams, Steve. *Programming the 68000.* Berkeley: Sybex, 1985.

Williams, Steve. *Programming the Macintosh in Assembly Language.* Berkeley: Sybex, 1986.

Wilson, David A.; Rosenstein, Larry S.; and Shafer, Dan. *Programming with MacApp.* Reading, MA: Addison-Wesley, 1990.

Index

Other Books Available in the Macintosh Inside Out series

- **Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  Here is the information you need to understand and use the power of MacApp, Apple Computer, Inc.'s official development environment for the Macintosh. The book discusses object-oriented concepts, using MPW with MacApp, the MacApp class library, and creating the Macintosh user interface. All examples are in Apple's Object Pascal language.
  576 pages, paperback
  $24.95, book alone, order number 09784
  $34.95, book/disk, order number 55062

- **C++ Programming with MacApp®**
  David A. Wilson, Larry S. Rosenstein, Dan Shafer
  In this book you will find information on using MacApp with C++, the up-and-coming language for Macintosh development. The book covers object-oriented techniques, MPW, and the MacApp class libraries. All program examples are in C++.
  600 pages, paperback
  $24.95, book alone, order number 57020
  $34.95, book/disk, order number 57021

- **Elements of C++ Macintosh® Programming**
  Dan Weston
  Macintosh programmers will learn just what they need to take the step from C to C++ programming, the future of Macintosh development. The book covers the basics and then teaches how to design practical programs with C++.
  464 pages, paperback
  $22.95, order number 55025

- **ResEdit™ Complete**
  Peter Alley and Carolyn Strange
  This book/disk package contains the actual ResEdit software along with a complete guide to using it. The book shows you how to customize your desktop and then moves on to cover more advanced topics such as creating standard resources, designing templates, and writing your own resource editor.
  560 pages, paperback
  $29.95 book/disk, order number 55075

- **The Complete Book of HyperTalk® 2**
  Dan Shafer
  This hands-on guide covers HyperTalk 2, with its greatly expanded features and capabilities. It offers practical information on commands, operators, and functions as well as detailed explanations of XCMDs, dialog boxes, menus, communications, and stack design. You'll also find plenty of tips and dozens of ready-to-use scripts.
  480 pages, paperback
  $24.95, order number 57082

Order Number	Quantity	Price	Total
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____
________	_____	____	_____
TOTAL ORDER			_____

Shipping and state sales tax will be added automatically.

Credit card orders only please.

Offer good in USA only. Prices and availability subject to change without notice.

Name ______________________________

Address ______________________________

____

City/State/Zip ______________________________

Signature (required) ______________________________

__Visa __MasterCard __AmEx

Account # ____________________ Exp. Date ________

Addison-Wesley Publishing Company
Order Department
Route 128
Reading, MA 01867
To order by phone, call (800) 477-2226

By MARK ANDREWS

Programmer's Guide to MPW®, Volume I:

EXPLORING THE MACINTOSH® PROGRAMMER'S WORKSHOP

Macintosh Inside Out

Special Characters Used in the MPW Command Language

Arranged by Category*

CATEGORY	CHAR.	PRESS	MEANING	USAGE	EXAMPLE	TRANSLATION
Comment	#	#	Characters between # and terminator are interpreted as a comment	#s	# This won't work	String following # is interpreted as a comment
Delimiter	"	"	Delimits a string in which each character is taken literally, except for ∂, {}, and `	"s"	Echo "{MPW}" >> "{Target}"	Echo the contents of the shell variable {MPW} to the target window
Delimiter	'	'	Delimits a string in which all characters are taken literally	's'	Echo '{MPW}' >> "{Target}"	Echo the string "{MPW}" to the target window
Delimiter	(	(	Delimits a group of characters that form a pattern; groups commands	(p)	Find /("*")+/	Select a group of one or more asterisks
Delimiter	)	)	Delimits a group of characters that form a pattern; groups commands	(p)	Find /("*")+/	Select a group of one or more asterisks
Delimiter	/	/	Searches forward and select regular expression	/r/	Find /delta/	Search forward and select the word "delta"
Delimiter	»	Option-Shift-\	Delimits number standing for number of occurrences	«n»	Find /[∂t]«2»/	Select exactly two tabs
Delimiter	»	Option-Shift-\	Delimits number standing for at least n occurrences	«n,»	Find /[∂t]«2,»/	Select two or more tabs
Delimiter	»	Option-Shift-\	Delimits number standing for n to n occurrences	«n1,n2»	Find /[∂t]«2,4»/	Select two to four tabs
Delimiter	«	Option-\	Delimits number standing for number of occurrences	«n»	Find /[∂t]«2»/	Select exactly two tabs
Delimiter	«	Option-\	Delimits number standing for at least n occurrences	«n,»	Find /[∂t]«2,»/	Select two or more tabs
Delimiter	«	Option-\	Delimits number standing for n to n occurrences	«n1,n2»	Find /[∂t]«2,4»/	Select two to four tabs
Delimiter	[	[	Delimits a pattern	[...]	Find /[A-F]/	Search for any character in the set A-F
Delimiter	\	\	Searches backwards and selects regular expression	\r\	Find \alpha\	Search backward and select the word "alpha"
Delimiter	]	]	Delimits a pattern	[...]	Find /[A-F]/	Search for any character in the set A-F
Delimiter	`	`	Send output of command c2 to command c1 for processing	c1 `c2`	Echo `Files -t TEXT`	Files command sends its output to Echo command, which prints the output on the screen
Delimiter	{	{	Delimits variable v	{v}	Echo "{MPW}"	Echo contents of shell variable {MPW}
Delimiter	}	}	Delimits variable v	{v}	Echo "{MPW}"	Echo contents of shell variable {MPW}
Escape	∂	Option-D	Return	∂n	Echo ∂n	Echo a return
Escape	∂	Option-D	Tab	∂t	Echo ∂t	Echo a tab
Escape	∂	Option-D	Form feed	∂f	Echo ∂f	Echo a form feed
Escape	∂	Option-D	Defeats the meaning of the special character that follows it	∂¬	Echo ∂¬	Output: ¬
Filename operator	*	*	Matches zero or more occurrences of the preceding character or character list	c*	X*	Match zero or more occurrences of the character X
Filename operator	?	?	Matches any single character in a file name	?	Source.?	Match any file that is named Source and has a one-character extension
Filename operator	?*	?* (same as ≈)	Matches any number of any characters in a file name	?*	?*.c	Match any file name with the extension ".c"
Filename operator	¬	Option-L	Matches any character not in the list	[¬list]	[¬A-F]	Match any character that is not in the set A-F
Filename operator	»	Option-Shift-\	Delimits number standing for number of occurrences	«n»	[X]«2»	Match two occurrences of the character X
Filename operator	«	Option-\	Delimits number standing for number of occurrences	«n»	[X]«2»	Match two occurrences of the character X
Filename operator	[	[	Delimits a pattern	[...]	[A-F]	Match any character in the set A-F
Filename operator	]	]	Delimits a pattern	[...]	[A-F]	Match any character in the set A-F
Filename operator	≈	≈	Matches any number of any characters in a file name	≈	≈.c	Match any file name with the extension ".c"
Filename operator	+	+	Matches one or more occurrences of the preceding character or characters	c+	X+	Match one or more occurrences of the character X
Line continuation	∂	Option-D	If ∂ stands alone at end of a line, MPW joins line to next line, ignoring return	∂	(First line:) Echo "How are ∂ (Second line:) you today?"	Output: How are you today?
Make	"	"	Delimits a string in which each character is taken literally, except for ∂, {}, and `	"s"	"{Libraries}" Runtime.o	The C runtime libraries
Make	#	#	Characters between # and terminator are interpreted as a comment	#s	### Dependency rules ###	String following # is interpreted as a comment
Make	'	'	Delimits a string in which all characters are taken literally	's'	'{Libraries}' Runtime.o	The runtime libraries
Make	∂	Option-D	If ∂ stands alone at end of a line, MPW joins line to next line, ignoring return	∂	(First line:) Sample ƒƒ Sample.p.o ∂ (Second line:) Sample.r	Output: Sample ƒƒ Sample.p.o Sample.r (all on one line)
Make	ƒ	Option-F	File f1 depends on file f2	f1 ƒ f1	Sample.p.o ƒ Sample.p	File Sample.p.o depends on file Sample.p
Make	ƒƒ	Option-F	File f1 depends on file f2, and f2 has its own build commands	f1 ƒƒ f1	Sample ƒƒ Sample.p.o	File Sample depends on file Sample. p.o, and Sample p.o. has its own set of build commands
Menus	!	!	Marks menu item with specified character	!c	!√	Mark menu item with a check mark
Menus	(	(	Disables menu item	(	(-	Place a dimmed horizontal line in menu list
Menus	-	-	Prints a horizontal line separating menu items	-	(-	Place a dimmed horizontal line in menu list
Menus	/	/	Associates a menu item with keyboard equivalent c	/c	/M	Assign control-M to be menu item's keyboard equivalent
Menus	<	<	Sets character style of a menu item (bold, italics, underlined, outline, or shadow)	<[BIUOS]	<B	Set character style of menu item to bold
Menus	^	^	Followed by an icon number, marks menu item with specified icon	^n	^2	Mark the menu item with icon no. 2 (in a resource fork)
No. prefix	$	$	Precedes hexadecimal number (same as 0x)	$[0-9A-Fa-f]+	Evaluate $9EFF + $9E	Output: 40861
No. prefix	0	0 (zero)	Precedes octal number	0[0-7]+	Evaluate 054 + 030	Output: 68
No. prefix	0b	0b	Precedes binary number	0b[0-1]+	Evaluate 0b11 - 0b01	Output: 2
No. prefix	0x	0x	Precedes hexadecimal number (same as $)	$[0-9A-Fa-f]	Evaluate $9EFF + $9E	Output: 40861
Operator	!	!	Not (same as NOT)	!n	Evaluate !0	Output: 1
Operator	<>	!= (same as !=, ≠)	True if n1 is not equal to n2	n1 <> n2	Evaluate 2 <> 3	Output: 1
Operator	!=	!= (same as <>, ≠)	True if n1 is not equal to n2	n1 != n2	Evaluate 2 != 3	Output: 1
Operator	%	% (same as MOD)	Returns mod n2	n1 % n2	Evaluate 25 % 4	Output: 1
Operator	&	&	Bitwise AND	n1 & n2	Evaluate 0b0001 & 0b0011	Output: 1
Operator	&&	&&	Logical AND	n1 && n2	Evaluate 1 && 1	Output: 1
Operator	*	*	Multiplies n1 by n2	n1 * n2	Evaluate 3 * 3	Output: 9
Operator	+	+	Adds n1 to n2	n1 + n2	Evaluate 1 + 1	Output: 2
Operator	-	-	Subtracts n1 from n2	n2 - n1	Evaluate 33 - 32	Output: 1
Operator	<	<	True if n1 is less than n2	n1 < n2	Evaluate 2 < 3	Output: 1
Operator	<<	<<	Shifts n1 left arithmetically n2 times	n1 << n2	Evaluate 0b0001 << 1	Output: 2
Operator	<=	<=	True if n1 is less than or equal to n2	n1 <= n2	Evaluate 2 <= 3	Output: 1
Operator	<=	<= (same as ≤)	True if n1 is less than or equal to n2	n1 <= n2	Evaluate 2 <= 3	Output: 1
Operator	==	==	True if n1 equals n2	n1 == n2	Evaluate 2 == 3	Output: 0
Operator	>=	>= (same as ≥)	True if n1 is greater than or equal to n2	n1 >= n2	Evaluate 3 >= 2	Output: 1
Operator	>>	>>	Shifts n1 right logically n2 times	n1 >> n2	Evaluate 0b0010 >> 1	Output: 2
Operator	DIV	DIV (same as ÷)	Divides n1 by n2	n1 DIV n2	Evaluate 25 DIV 5	Output: 5
Operator	MOD	MOD (Same as %)	Returns mod n2	n1 MOD n2	Evaluate 25 MOD 4	Output: 1
Operator	NOT	NOT	Not (same as !)	NOT n	Evaluate NOT 0	Output: 1
Operator	÷	Option-/ (same as DIV)	Divides n1 by n2	n1 ÷ n2	Evaluate 25 ÷ 5	Output: 5
Operator	≤	Option-< (same as <=)	True if n1 is less than or equal to n2	n1 ≤ n2	Evaluate 2 ≤ 3	Output: 1
Operator	≠	Option-= (same as !=, <>)	True if n1 is not equal to n2	n1 ≠ n2	Evaluate 2 ≠ 3	1
Operator	≥	Option-> (same as >=)	True if n1 is greater than or equal to n2	n1 ≥ n2	Evaluate 3 ≥ 2	Output: 1
Operator	^	^	Bitwise XOR	n1 ^ n2	Evaluate 0b0001 ^ 0b0011	Output: 2
Operator	|	|	Bitwise OR	n1 | n2	Evaluate 0b0001 | 0b0011	Output: 3
Operator	||	||	Logical OR	n1 || n2	Evaluate 1 || 0	Output: 1
Operator	~	~	Negates number	~n	Evaluate ~ 4	Output: -5
Redirection	<	<	Standard input is taken from file name f	<f	Alert < Errors	Display an alert dialog containing the contents of the file Errors
Redirection	>	>	Redirects standard output, replacing contents of file f	>f	Echo "{Status}" > Errors	Write contents of shell variable {Status} to file Errors, replacing its previous contents
Redirection	>>	>>	Redirects standard output, appending it to contents of file f	>>f	Echo "{Status}" >> Errors	Append contents of shell variable {Status} to the end of file Errors.
Redirection	≥	Option->	Redirects diagnostics, replacing contents of file f	≥f	(Files ≈.p ||) ≥ Errors	List filenames that end in ".p". Send diagnostics to file Errors, replacing its contents
Redirection	≥≥	Option->	Redirects and appends diagnostics to file f	≥≥f	(Files ≈.p ||) ≥ ≥Errors	List filenames that end in ".p". Append diagnostics to end of file Errors
Redirection	Σ	Option-W	Redirects both standard output and diagnostics to file f, replacing its contents	Σf	(Files ≈.p ||) Σ Temp	List filenames ending in ".p". Send output, diagnostics to file Temp, replacing its contents
Redirection	ΣΣ	Option-W	Redirects and appends both standard output and diagnostics to file f	ΣΣf	(Files ≈.p ||) ΣΣ Temp	List filenames ending in ".p". Append output and diagnostics to file Temp
Regular expression operator	!~	!~	True if s1 is not equal to s2	"s1" !~ /s2/	Evaluate "alpha" !~ /beta/	Output: 1
Regular expression operator	*	*	Selects zero or more occurrences of regular expression	r*	Find /("*")+'/' [∂r∂t]*/	Select a group of one or more asterisks followed by a slash bar and 0 or more white spaces
Regular expression operator	+	+	Selects one or more occurrences of regular expression	r+	Find /("*")+'/'/	Select a group of one or more asterisks
Regular expression operator	+	+	Matches one or more occurrences of the preceding character or characters	r+	X+'	Match one or more occurrences of the character X
Regular expression operator	-	-	Stands for range of characters between c1 and c2	c1-c2	Find /[A-Za-z]+∂n/	Select any word made up of upper- and lowercase letters that appears at the end of a line
Regular expression operator	=~	=~	True if s1 is equal to s2	"s1" =~ /s2/	Evaluate "beta" =~ /beta/	Output: 1
Regular expression operator	:	Colon	All text between (two selections)	s:s	Find •:∞	Select (highlight) all text in file
Regular expression operator	∞	Option-5	(With command that takes a -c option): Repeats command to end of file	cmd -c ∞	Replace -c ∞ /123/ 456	Replace string "123" with string "456" every time it appears in target window
Regular expression operator	∞	Option-5	Selects regular expression at the end of a line	r∞	Find /arlie∞/	Select the letters "arlie" at the end of a line
Regular expression operator	•	Option-8	Selects regular expression at the beginning of a line	•r	Find /•ch/	Select the letters "ch" at the beginning of a line
Regular expression operator	…	Option-;	Executes Commando command, invokes Commando dialog for command c	c…	TileWindows…	Invoke TileWindows Commando
Regular expression operator	¬	Option-L	Any character not in the list	[¬list]	Replace -c ∞ /[¬A-Za-z∂n" "]/ "*"	Replaces all characters except A-Z, a-z, returns and spaces with asterisks
Regular expression operator	®	Option-R	Tags regular expression with a number (range: 1-9)	r®n	Replace /([a-zA-Z]+)®1[ ]+([a-zA-Z]+)®2/ '®2 ®1'	Reverse the order of two words separated by one or more spaces
Selection	!	!	Selects the line that is n lines after end of current selection	!n	Find !3	Select the third line after the current selection
Selection	!	!	Places insertion point n characters after regular expression	r!n	Find /alpha/!3	Place insertion point three characters after the word "alpha"
Selection	¡	Option-!	Places insertion point n lines before start of current selection	¡n	Find ¡3	Place insertion point three lines before start of current selection
Selection	∞	Option-5	Selects end of file	∞	Find ∞	Place insertion point after last character in file
Selection	§	Option-6	Current selection	§	Copy §	Copy the current selection (highlighted text) to the Clipboard
Selection	•	Option-8	Selects beginning of file or line	•	Find •	Place insertion point before first character in file or file
Selection	Δ	Option-J	Places insertion point before first character in regular expression	Δr	Find Δ/charlie/	Place insertion point before first character in the word "charlie"
Selection	Δ	Option-J	Places insertion point after last character of regular expression	rΔ	Find /charlie/Δ	Place insertion point after last character of the word "charlie"
Terminator	&&	&&	Executes command c2 if command c1 succeeds	c1 && c2	Find /charlie/ && Echo Found!	If string "charlie" is found, MPW echoes, "Found!"
Terminator	;	;	Treats commands on the same line as if they were on different lines	c ; c	Echo hello ; Echo goodbye	Output: (First line:) Hello (Second line:) Goodbye
Terminator	Return	Ret	Ends command	c (r)	Echo Hello(r)	Output: Hello
Terminator	|	|	Pipes output of command c1 to input of c2	c1 | c2	Files | Count -l	Files pipes a list of files to Count, which prints the list on the screen
Terminator	||	||	Executes command c2 if command c1 fails	c1 || c2	Find /zebra/ || Echo Sorry!	Searches for string "zebra" and echoes "Sorry!" if search fails
Whitespace	Space	Space	Separates words	w w	Echo Hello	Output: Hello
Whitespace	Tab	Tab	Separates words	w w	Echo Hello	Output: Hello
Wildcard	?	?	Matches any single character in a string	?	Find /Bar?/	Select any four-character word that begins with "Bar"
Wildcard	?*	?*	Matches any number of occurrences of any character (same as ≈)	chars?*	Find /Mar?*/	Select any word that begins with "Mar"
Wildcard	≈	Option-X	Matches any number of any characters in a string	≈	Find /Mar≈/	Select any word that begins with "Mar"

*The information in this table is listed in character order in the text.

Macintosh Programming

Macintosh Inside Out

> $26.95 FPT USA
> $34.95 CANADA

Programmer's Guide to MPW®, Volume I

MARK ANDREWS

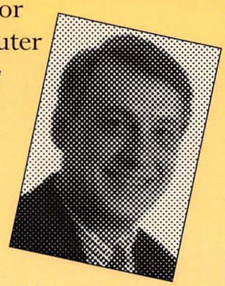
Learn the secrets to unlocking the power of MPW® version 3.2, the newest release of the Macintosh® Programmer's Workshop. MPW is Apple® Computer, Inc.'s official integrated software development system for the Macintosh, and this definitive guide will provide you with everything you need to create and design effective and efficient Macintosh applications using MPW and System 7.0.

Programmer's Guide to MPW, Volume I first covers the fundamentals of MPW, including the MPW Editor, the command language and menu structure, dialogs, and scripting. The book then builds on these skills to discuss more advanced programming techniques dealing with the Macintosh Event Manager, Resource Manager, and Memory Manager. In the final section of the book, you will build a fully functional application which can be used as a template to create your own programs.

You will also learn how to:

- Customize menus
- Make calls to the Macintosh Toolbox and operating system from Pascal, C, and assembly language
- Create new MPW commands and scripts for specialized tasks
- Create object code libraries you can call from your programs
- Compile and link application programs and much more.

Appendices contain the complete MPW command set and all source code listings in the book. The book also features an easy-reference tear-out chart presenting the full set of special characters. This thorough coverage of MPW tools and techniques makes **Programmer's Guide to MPW, Volume I** an essential guide for all Macintosh programmers.



Mark Andrews is the author of more than a dozen computer books including *Programming the Apple IIGS® in Assembly Language and C*. He also worked as an independent consultant and Quality Engineer for Apple Computer, Inc., during the development of MPW 3.2 and System 7.0. He is currently a senior technical writer at Oracle Corporation.

52695>

9 780201 570113

ISBN 0-201-57011-4

57011

Cover design by Ronn Campisi

Addison-Wesley Publishing Company, Inc.