M A C I N T O S H
P R O G R A M M E R ' S
W O R K S H O P

Introduction to MPW

Second Edition

Developer Press

Contents

Figures, Tables, and Listings

About This Guide

This guide, *Introduction to MPW*, second edition, contains introductory material you need to know before you can use MPW to create Macintosh software. The material in this guide is presented in a narrative style, interspersed with tutorial examples that guide you through using the MPW Shell.

All of the tutorials are useful, but none of them is required; the chapters will make sense to you even if you choose not to do the tutorials. To save you learning time, this guide presents the information in a sequential manner so that each chapter builds on the previous one.

How to Use This Guide

After you read the Preface, you should read this guide chapter by chapter, so that you are familiar with the information in earlier chapters before you read the later chapters. Although simply reading this guide gives you a feel for using MPW, it is recommended that you complete the tutorials.

This guide includes the following chapters:

- "Getting Started" contains the information you need to begin using the MPW Shell and to build the MPW sample applications.
- "Using Commando Dialog Boxes" shows you how to enter commands by using dialog boxes.
- "Entering Commands" introduces you to the MPW Shell command language.
- "Working With Files and Directories" describes how to use the MPW Shell to create source files and how to use file and directory names in MPW Shell commands.
- "Editing With Commands" describes the rules and syntax of using MPW Shell commands to accomplish powerful editing functions.
- "Writing Scripts" introduces you to the techniques of combining MPW Shell commands into scripts.

In addition, this guide contains several useful appendixes:

- "The Rules of Using Quotation Marks" lists the rules of using quotation marks in MPW Shell commands.
- "Table of Special Characters and Operators" lists all of the special characters and operators in the MPW Shell command language and divides them into functional categories.
- "A Sample UserStartup•*name* File" provides a sample file you can use to customize your environment.

At the end of this guide, you'll find a glossary of useful terms, as well as an index.

Conventions Used in This Guide

This guide uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Important information, such as rules for using the MPW Shell command language, is presented in a special format so that you can scan it quickly.

Special Fonts

This manual uses the Letter Gothic typeface to represent sample code, elements of the command language, text you enter, or text MPW displays (`this is Letter Gothic`).

Key terms or concepts are shown in boldface and are defined in the Glossary (**this is boldface**).

Types of Notes

This guide contains several types of notes.

Note

A note like this one contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT

A note like this one contains information that is essential for an understanding of the main text. ▲

▲ **W A R N I N G**

Warnings like this indicate potential problems that you should be aware of as you use the MPW Shell. Failure to heed these warnings could result in system crashes or loss of data. ▲

QUOTING RULE #1

A note like this presents an important rule of using quotation marks in the MPW Shell command language. ◆

In addition, this guide contains many hands-on tutorials, which are marked with an icon in the left margin, like the one to the left of this paragraph.

Syntax Conventions

In addition, this guide uses certain syntax conventions to present MPW Shell commands. The syntax conventions used in this guide are as follows:

`literal`	Letter Gothic text indicates a word that must appear exactly as shown. You must also enter special symbols (∂, •, §, &, and so on) exactly as they are shown.
italics	Italics indicate a parameter that you must replace with anything that matches the parameter's definition.
[ ]	Brackets indicate that the enclosed item is optional.
...	Ellipses indicate that the preceding item can be repeated one or more times.
|	A vertical bar indicates an either/or choice.

For More Information

APDA is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

C H A P T E R 1

Getting Started

Contents

Welcome to MPW, the Macintosh Programmer's Workshop. To access the power of MPW, you will use the application known as the MPW Shell. The MPW Shell is the gateway that gives you access to the rest of MPW. From the MPW Shell, you can create source files, enter commands, compile programs, and use any of the MPW tools and utilities.

This chapter introduces you to the MPW Shell and describes the basic skills you need to acquire so that you can use MPW. Specifically, this chapter gives you a tour of the MPW folder and then describes how to

- start MPW
- use the Worksheet window to enter commands
- use the built-in help facility
- build a sample application

This chapter includes several hands-on tutorials. If you are new to MPW, especially if you are new to programming the Macintosh, read this chapter and do the tutorials. Otherwise, you can skip to the next chapter, "Using Commando Dialog Boxes."

Hardware and System Requirements

Before you read this chapter, you should install MPW. For installation instructions, you can refer to the file named MPW—Read Me First on the CD-ROM you received.

To install and run MPW, you need certain hardware and software, including:

- A Macintosh computer with at least a Motorola 68020, 68030, or 68040 microprocessor or any Power Macintosh.
- A hard disk with at least 40 megabytes of free space.
- At least 8 megabytes of memory. If you are developing large or complex applications, you will need more memory.
- System 7 or later.
- A CD-ROM drive to install MPW from the MPW Pro CD-ROM. (Though the CD-ROM drive is not required to run MPW, you might prefer to access the MPW documentation, technical notes, sample code, and the Online

References from the MPW Pro CD-ROM rather than copying them to your hard drive.)

Note
The MPW environment is subject to change. As a result, contents of the MPW Pro CD-ROM can vary from release to release. Do not be concerned if your software configuration is not identical to the one described in this book. The document "About MPW Pro" on the MPW Pro CD-ROM describes the contents of your particular release and provides installation instructions. ◆

A Tour of the MPW Folder

If you have installed MPW correctly on your hard disk, you can find its files in a folder named MPW. Your MPW folder contains many items, some of which are standard to MPW, and some of which are specific to your MPW environment. When you first install MPW, your MPW folder contains at least the items shown in Figure 1-1.

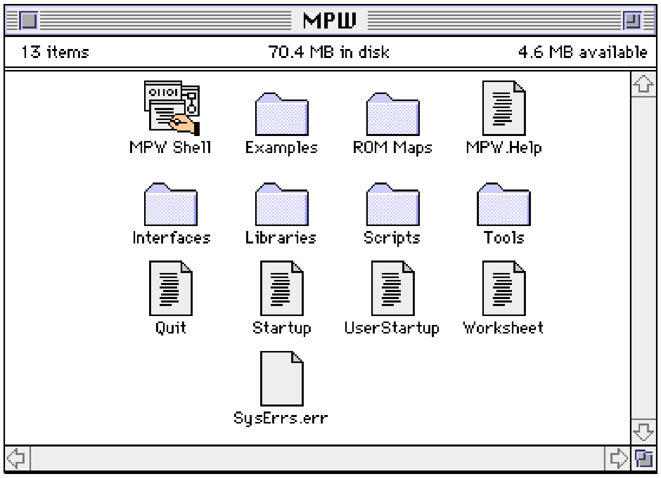
The first item you see in the MPW folder is the MPW Shell icon, which you will use to launch the MPW Shell and enter the MPW development environment. The MPW Shell allows you to create and edit source files and to enter commands that are executed by a powerful command interpreter.

In the MPW Shell, there are three types of commands. You enter all three types of commands in the same way—by typing the command in a window, then pressing Enter. However, the three types differ slightly in how they are executed.

When you enter a command, it might be

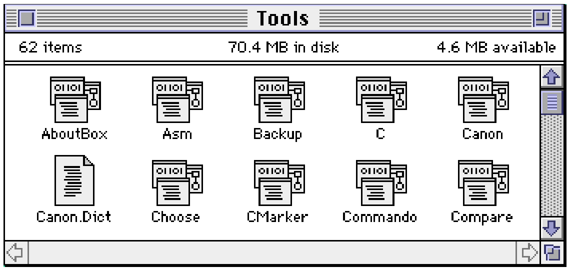
- A **built-in command**, which is part of the MPW Shell.
- A **tool**, which is a small executable program stored in the Tools folder.
- A **script**, which is a file that contains a series of commands.

Figure 1-1 A sample MPW folder



The Tools folder contains dozens of tools, a few of which are shown in Figure 1-2.

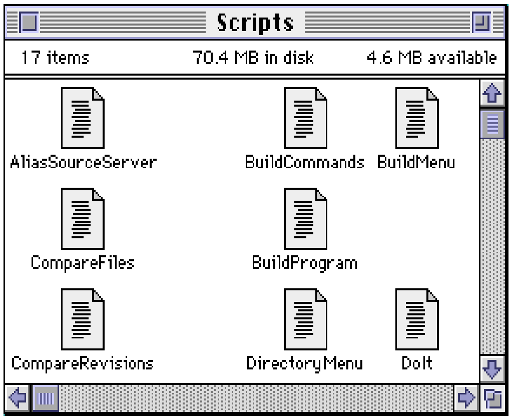
Figure 1-2 Part of the Tools folder



To use a tool, you must enter a command in an MPW Shell window. The command you enter is the same as the tool name, sometimes with additional options and parameters. You cannot use a tool by double-clicking its icon.

The Scripts folder (Figure 1-3) contains some MPW Shell scripts.

Figure 1-3 Part of the Scripts folder



Scripts can speed and automate many of the tasks you perform with MPW. You execute a script by entering its name—sometimes with additional information, sometimes without—in an MPW Shell window.

In this guide, the term *command* is used to mean a command line you type in an MPW Shell window, regardless of whether it is executed by a built-in command, a script, or a tool.

The contents of the Examples, Libraries, and Interfaces folders vary according to which version of MPW you have installed. The Examples folder stores sample applications written for different MPW programming languages and runtime environments.

If you open the Examples folder (Figure 1-4), you see the sample applications you have installed. Before you experiment with the sample applications, you should read and complete the section "Tutorial—Building and Running SillyBalls" later in this chapter.

The MPW folder contains two other important folders, Libraries and Interfaces. The Libraries folder contains program libraries that you can link with your source files, and the Interfaces folder contains include files that define the interfaces to routines in the libraries and Macintosh Toolbox. Within both the Libraries and Interfaces folders are folders for the different MPW languages and runtime environments, as shown in Figure 1-5 and Figure 1-6 on page 1-9.

Figure 1-4 The Examples and CExamples folders

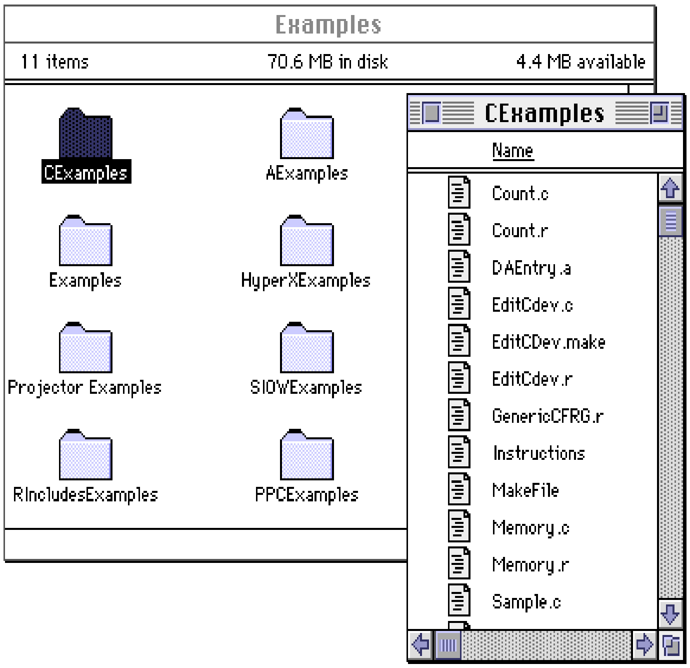


Figure 1-5 The Libraries and CLibraries folders

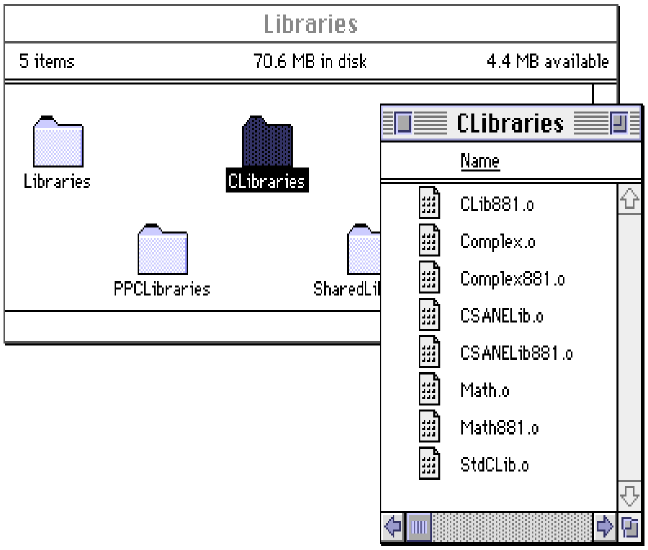
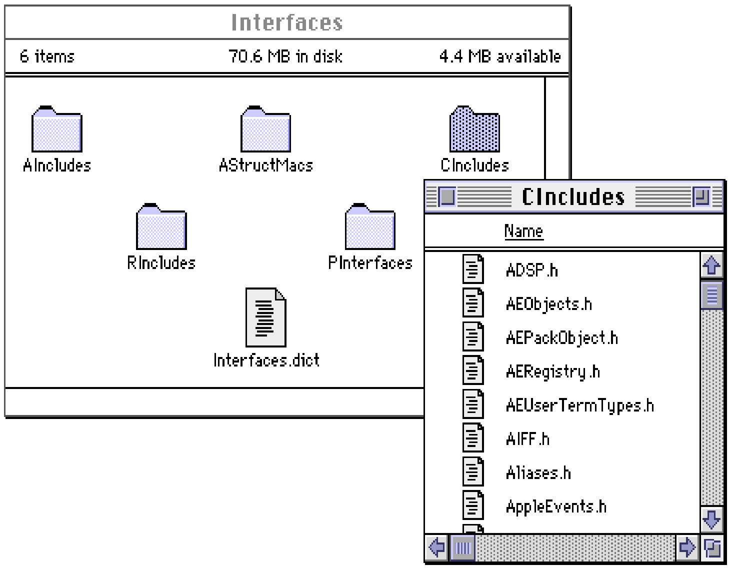


Figure 1-6 The Interfaces and CIncludes folders



The MPW folder also contains a set of documents with names like Startup, UserStartup, and Worksheet that the MPW Shell uses as you perform everyday tasks.

Finally, the MPW folder includes an `MPW.help` file that the built-in MPW help system uses when you want to look up information about MPW and its commands.

Starting MPW

When you start MPW, you launch the MPW Shell and enter the MPW development environment, where you have access to all of MPW's tools, scripts, compilers, and libraries. To start MPW, open the MPW folder and double-click the MPW Shell icon (Figure 1-7) or any MPW document (Figure 1-8).

Figure 1-7 The MPW Shell icon

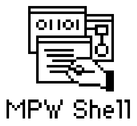


Figure 1-8 An MPW document icon

The Worksheet Window

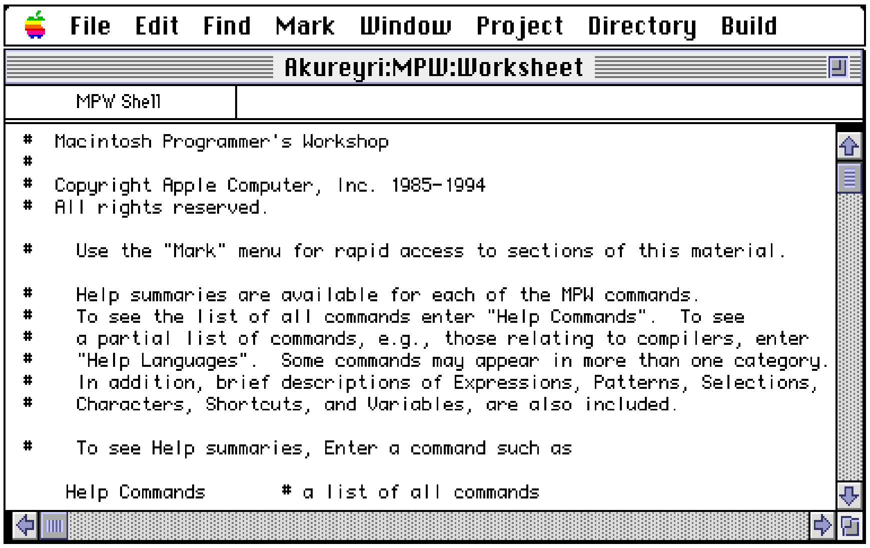
When you start MPW, it reads its startup files, initializes itself, builds its menu bar, and then displays a window called the Worksheet window (Figure 1-9 on page 1-11).

The Worksheet window has some special characteristics. Unlike most Macintosh windows, it is not empty when you first start MPW; it contains some useful help text that describes how to use the MPW Shell's help command.

Notice that the window name is different from window names in other Macintosh applications. In MPW Shell windows, window names include a hard disk name, one or more folder names, and a filename. Each part of the name is separated from the next by a colon. The Worksheet window shown in Figure 1-9 shows that the MPW Shell is stored on the hard disk named graphics, in the folder named MPW. If the window name is long, it is truncated and displayed with an ellipsis in the middle, like this name:

```
graphics...Worksheet
```

Figure 1-9 The Worksheet window and the MPW Shell menu bar



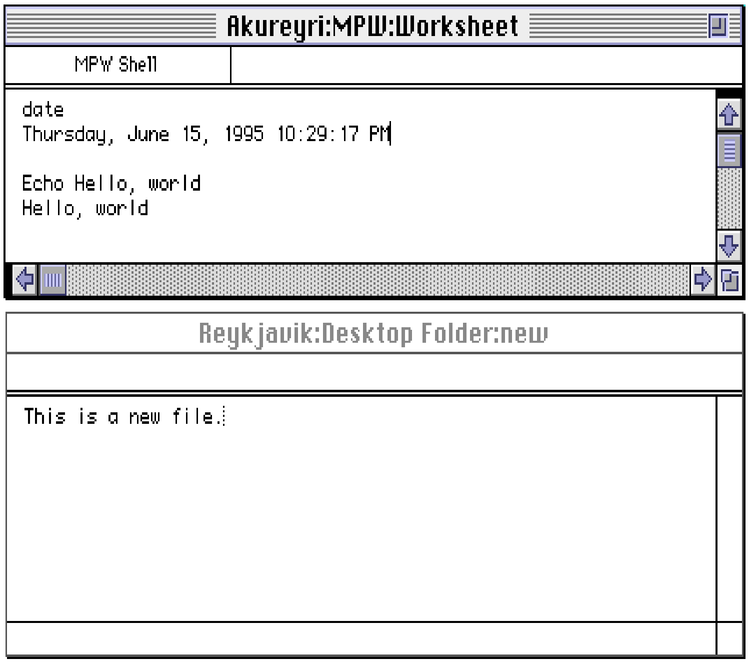
```
#  Macintosh Programmer's Workshop
#
#  Copyright Apple Computer, Inc. 1985-1994
#  All rights reserved.

#    Use the "Mark" menu for rapid access to sections of this material.

#    Help summaries are available for each of the MPW commands.
#    To see the list of all commands enter "Help Commands".  To see
#    a partial list of commands, e.g., those relating to compilers, enter
#    "Help Languages".  Some commands may appear in more than one category.
#    In addition, brief descriptions of Expressions, Patterns, Selections,
#    Characters, Shortcuts, and Variables, are also included.

#    To see Help summaries, Enter a command such as

    Help Commands       # a list of all commands
```

Note
In the rest of this guide, the term **directory** is used to refer to a folder, and the term **volume** is used to refer to a hard disk. ◆

You'll also notice that the Worksheet window has no close box in its upper-left corner. That's because the **Worksheet window** is always displayed when the MPW Shell is running. This makes the Worksheet window a convenient place to enter commands and receive output from the MPW Shell.

When you enter a command in the Worksheet window, it flashes briefly in the **status panel** in the upper-left corner of the window. The status panel displays the name of the command that the MPW Shell is currently executing. When the MPW Shell is not executing a command, the status panel displays `MPW Shell`. You can have other windows open along with the Worksheet window, as shown in Figure 1-10.

Figure 1-10 Working with several windows open



```
date
Thursday, June 15, 1995 10:29:17 PM

Echo Hello, world
Hello, world
```

```
This is a new file.
```

When you click in a window, it becomes selected and is called the **active window.** You can tell that a window is selected when you see the dark horizontal lines across the window's title bar. If you have arranged your windows so that several windows overlap or are on top of each other, the active window moves to the front.

In Figure 1-10, you can see that the Worksheet window is the active window, because it is selected. The active window is where the MPW Shell displays its output and error messages, unless you specify otherwise. All other windows on your screen except one are known as **inactive windows.** (One of the inactive windows has some special characteristics and is called a **target window.** Target windows are described in the chapter "Editing With Commands" later in this guide.)

What Happens When You Start MPW

When you start the MPW Shell, the first five menus in the menu bar (File, Edit, Find, Mark, and Window) appear right away, but the last three (Project, Directory, and Build) take a few seconds to appear. This is because the MPW Shell starts and then executes the following startup files:

■ Startup, which initializes values that affect the MPW environment, such as the default format of text you type in windows

■ UserStartup, which builds the Project, Directory, and Build menus

As each command in these startup files is executed, it flashes briefly in the status panel in the Worksheet window.

IMPORTANT

Do not make changes to the Startup or UserStartup files. If you introduce bugs into any of these files, you could cause problems. And even if you don't, you would overwrite your changes when you reinstall or update MPW because the standard startup files will be restored. To set values that you want initialized each time you start MPW, you can create a separate file, with a name in the form UserStartup•*name*. You can find a sample UserStartup•*name* file in the appendix "A Sample UserStartup•*name* File." ▲

Moving the MPW Shell Icon

You may prefer to store an MPW Shell icon somewhere other than the MPW directory; for example, on your desktop. If so, make an alias for the icon by returning to the Finder, selecting the MPW Shell icon, and then choosing Make Alias from the File menu. Then, you can move the alias—rather than the original icon—to the new location.

Note

In the rest of this guide, aliases for files, directories, or volumes that you create from the Finder are called **Finder aliases**. The types of Finder aliases are explained in more detail in the chapter "Working With Files and Directories." ◆

Using the Worksheet Window

Because the Worksheet window is always displayed, it is an excellent place to enter commands. You can have a number of MPW windows displayed at once, and you can enter commands and receive messages from the MPW Shell in any of them. However, it is usually best to use the Worksheet window for entering commands and receiving responses and to use other windows for creating documents such as source files.

How to Enter a Command

To enter a command in the Worksheet window, you scroll to any blank line, place the insertion point on the line, and then type the command. Then, to send the command to the command interpreter to be executed, leave the insertion point on the same line as the command, and do any of the following:

- Press Enter.
- Press Command-Return.
- Click the status panel.

If you press Enter, be sure to use the Enter key at the far right of the keyboard—not the Return key.

By default, MPW Shell commands are not case sensitive. For example, instead of entering the command `Date`, you could enter `date` or `DATE` and get the same result.

Note
Throughout this guide, the term *enter a command* is used to mean type a command, then use the method you prefer to send it to the command interpreter. Also, throughout this guide, commands begin with an initial uppercase letter followed by lowercase letters. As you use this guide, remember that you can enter commands with any combination of uppercase and lowercase letters, unless you make them case sensitive, as explained in the chapter "Working With Files and Directories." ◆

You can stop any command while it is executing by pressing Command-period, the standard Macintosh keystroke combination for terminating commands. This is a simple skill, but an important one.

Whenever you are not sure what the MPW Shell is doing, you can check the status panel in the Worksheet window. If you see a command's name displayed, you know that the MPW Shell is busy executing the command. To terminate the command, press Command-period.

Tutorial—Getting Started With Commands

In this tutorial, you will

- start the MPW Shell and observe its startup process
- enter some simple commands in the Worksheet window
- execute a command from a dialog box, rather than by typing it
- enter a command with a comment

This tutorial provides only an introduction to the MPW Shell command language. To learn to use the command language, you also need to read the chapter "Entering Commands."

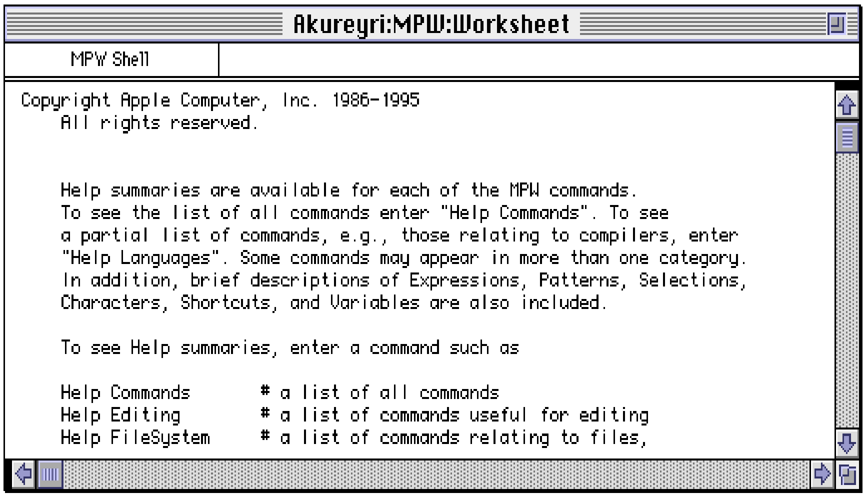
To start the MPW Shell and enter commands, follow these steps:

1. **Open the MPW directory, then double-click the MPW Shell icon.**

   When the Worksheet window appears (Figure 1-11), watch the status panel. Notice that as commands flash in the status panel, the Project, Directory, and Build menus are added to the menu bar.

   When the startup process is complete, the Worksheet window is ready for you to enter commands.

Figure 1-11 The Worksheet window



Akureyri:MPW:Worksheet

MPW Shell

```
Copyright Apple Computer, Inc. 1986-1995
    All rights reserved.


    Help summaries are available for each of the MPW commands.
    To see the list of all commands enter "Help Commands". To see
    a partial list of commands, e.g., those relating to compilers, enter
    "Help Languages". Some commands may appear in more than one category.
    In addition, brief descriptions of Expressions, Patterns, Selections,
    Characters, Shortcuts, and Variables are also included.

    To see Help summaries, enter a command such as

    Help Commands       # a list of all commands
    Help Editing        # a list of commands useful for editing
    Help FileSystem     # a list of commands relating to files,
```

2. **Scroll to the bottom of the help text and place the insertion point on a blank line.**

   For now, it is not necessary to read the help text. However, you should read the next section and complete its tutorial, which describes how to use the help text to get online help.

3. **Type this command, then press Enter:**

   ```
   Date
   ```

   The MPW Shell executes the command and responds with the date and time, like this:

   ```
   Thursday, June 17, 1993 2:31:43 PM
   ```

4. **Now type this command, then press Enter:**

   ```
   date -t
   ```

   You have just entered the `Date` command with its `-t` option, which displays only the time. The MPW Shell responds with the time, like this:

   ```
   2:33:22 PM
   ```

   Notice that you entered the command in all lowercase letters.

5. **Type a third command, then press Enter:**

   ```
   Date -s
   ```

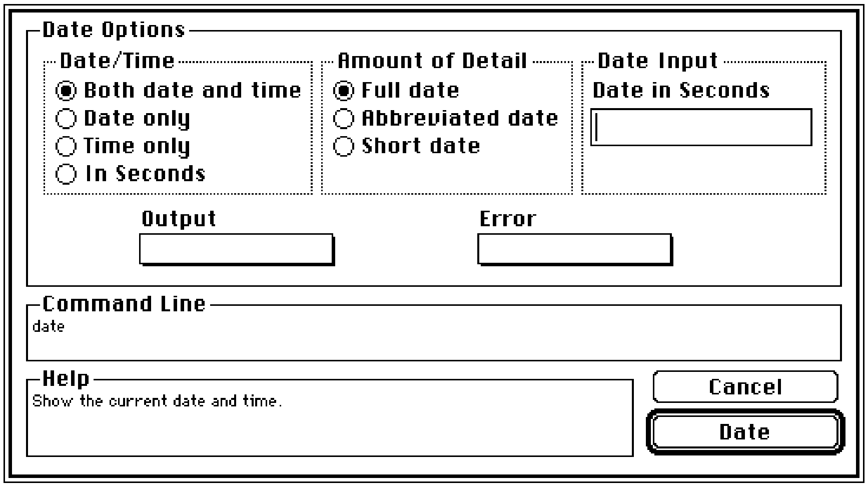
   The MPW Shell responds with the date and time in a shortened form, as in

   ```
   6/17/93 2:35:49 PM
   ```

6. **Type** `Date`**, then press Option-Enter, instead of Enter.**

   A dialog box that shows all the options available with the `Date` command appears (Figure 1-12).

Figure 1-12 Executing the Date command from a dialog box



Date/Time: [✓ Both date and time] Date only Time only In Seconds

Amount of Detail: [✓ Full date] Abbreviated date Short date

Date Input: Date in Seconds

Command Line

date

Help

Show the current date and time.

The dialog box, known as a **Commando dialog box,** allows you to execute the `Date` command with any of its options by clicking on buttons and boxes rather than by typing it in the Worksheet window.

Every MPW Shell command has a corresponding Commando dialog box. Commando dialog boxes are introduced here and explained in more detail in the next chapter.

Notice that the Command Line box (near the bottom of the Commando dialog box) contains the command

```
Date
```

7. **Click the "Short date" button in the Amount of Detail box.**

   The command in the Command Line box changes to

   ```
   Date -s
   ```

   which is the same command you entered in step 5.

8. **Click the Date button to execute the command.**

   The MPW Shell displays a short date and time:

   ```
   6/17/93 2:38:49 PM
   ```

   Notice that the date and time are in the same format as when you typed
   `Date -s`

9. **Type this command in the Worksheet window, then press Enter:**

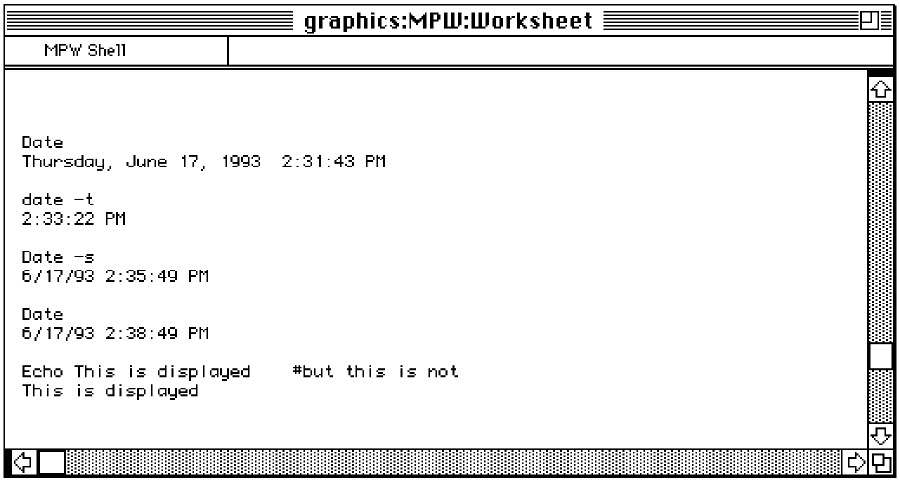
   ```
   Echo This is displayed      #but this is not
   ```

   The MPW Shell responds with

   ```
   This is displayed
   ```

   The text between the # character and the end of the line is a comment and is not executed as part of the command. By now, your Worksheet window should look something like the one in Figure 1-13.

Figure 1-13 Entering commands in the Worksheet window



```
Date
Thursday, June 17, 1993  2:31:43 PM

date -t
2:33:22 PM

Date -s
6/17/93 2:35:49 PM

Date
6/17/93 2:38:49 PM

Echo This is displayed    #but this is not
This is displayed
```

10. **Leave the MPW Shell running and the Worksheet window open so that you can do the remaining tutorials in this chapter.**

In this tutorial, you have learned the basic methods of entering commands in the Worksheet window and using Commando dialog boxes. Now you are prepared to experiment with any of the MPW Shell commands, including the commands that are part of the online help system. Remember that to enter a command, you type the full command, then press Enter; when you open a Commando dialog box, you type the command name, then press Option-Enter.

Commando dialog boxes are an excellent tool for experimenting with and learning the MPW Shell commands. Even experienced users find Commando dialog boxes invaluable for entering commands they do not use frequently or commands that have a complex syntax. Commando dialog boxes are described in more detail in the chapter "Using Commando Dialog Boxes."

Using the Online Help System

The MPW Shell has a built-in help system that describes its command language and the characters that have special meaning in commands. (The built-in online help system is part of the MPW shell, and is separate from the Macintosh Programmer's Toolbox Assistant help system, which provides extensive help on programming the Macintosh, and the Online Command Reference, which describes the MPW commands, tools, and scripts. You can find both these online help resources in the `Online Reference` folder on the CD-ROM.)

The built-in help is organized in a hierarchical fashion so that you can get help on many different topics, and on many different levels within a topic. To get online help, you use the `Help` command. For example, if you enter

```
Help
```

the MPW Shell displays a list of the topics included in the online help. The first few topics in the list are:

```
Help Commands      # a list of all commands
Help Editing       # a list of commands useful for editing
Help FileSystem    # a list of commands relating to files,
                   #   directories, and volumes
```

Each topic is an MPW Shell command. To get help on any of the topics, you can enter any of the commands on the left side of the list. The help commands, and the type of help they produce, are listed in Table 1-1.

Table 1-1 A list of topics in the online help

Command	What it displays
`Help`	A list of topics in the online help
`Help Characters`	A summary of other special characters used in the MPW Shell
`Help command`	The syntax of the command you specify
`Help Commands`	A list of all MPW Shell commands
`Help Editing`	A list of commands used in text editing
`Help Expressions`	A summary of the special characters used in arithmetic and logical expressions
`Help FileSystem`	A list of commands relating to files, directories, and volumes
`Help Languages`	A list of commands that compile and assemble programs
`Help Launch`	Information on launching applications from the MPW Shell.
`Help Miscellaneous`	A list of tools and utilities that can help you develop software
`Help Patterns`	A summary of the special characters used in expressions that match text
`Help PowerMacintosh`	A list of commands used only for Power Macintosh development.
`Help Projector`	A list of commands for using Projector, the MPW Shell's project management system
`Help Resources`	A list of commands useful when processing resources.
`Help Scripting`	A list of commands used within scripts
`Help Selections`	A summary of the special characters used to select a position in a text file
`Help Shortcuts`	A list of shortcuts you can use with the MPW Shell

Table 1-1 A list of topics in the online help (continued)

Command	**What it displays**
`Help System`	A list of commands that perform system functions
`Help Variables`	A summary of the internal variables the MPW Shell uses
`Help Window`	A list of commands that manage windows

Getting More Specific Help

To get help on any of the topics, enter any of the commands on the left side of the list. For example, if you enter

```
Help Commands
```

the MPW Shell displays a list of all its commands, with a brief description of each. The list is like this one, but longer:

```
Help AddMenu       # add a menu item
Help AddPane       # split the window into panes
Help Adjust        # adjust lines
```

To get further help on any of the commands, you can enter a command like

```
Help Adjust
```

adding the command's name after `Help`. The MPW Shell displays the syntax of the command you specified, with an explanation of each of the command's options, like this:

```
Adjust             # adjust lines
Adjust [-c count] [-l spaces] selection [window]
        -c count            # repeat the Adjust count times
        -l spaces           # shift lines by "spaces" spaces
```

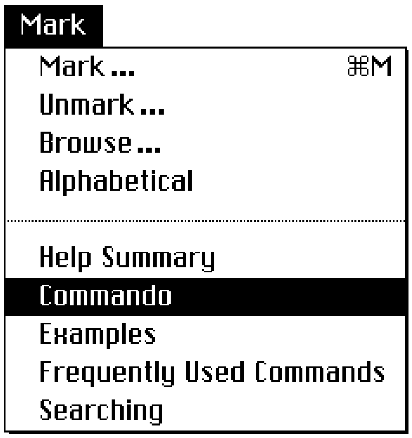
Once you know what the command's options are, you can enter it in the Worksheet window.

The Markers in the Help Text

The help text in the Worksheet window contains **markers** to help you move through it quickly. Even though the help text looks short, the markers are useful because as you work, you typically add text to the Worksheet window. (You can also add markers to your own files to identify routines, declarations, or other points of interest, as described in the chapter "Working With Files and Directories.")

To move through the help text by using markers, you must use the Mark menu (Figure 1-14).

Figure 1-14 The Mark menu, showing markers for the Worksheet window



When you choose a marker name from the bottom half of the menu, the text in the Worksheet window scrolls to that topic in the help text. For example, if you choose Commando from the menu, the Worksheet window scrolls to the beginning of the help text that describes Commando dialog boxes.

About the Special Characters

Many characters, known in MPW as **special characters,** have a special meaning when they are used in MPW Shell commands. If you enter the command

```
Help Characters
```

the MPW Shell displays a summary of some of the special characters you can use in commands. Notice the following lines in the list:

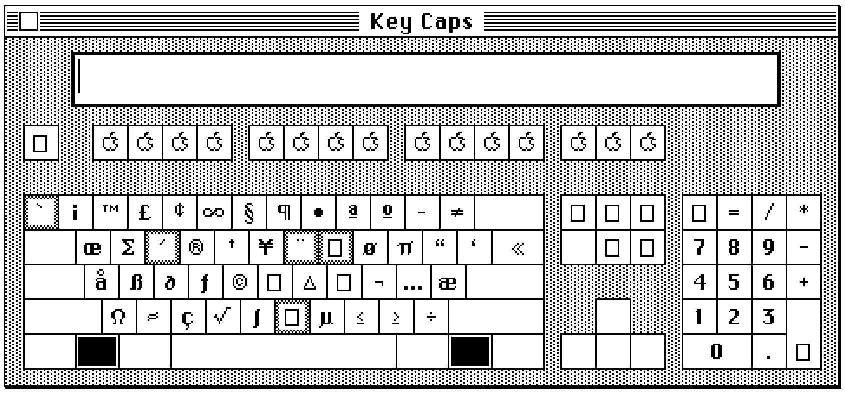
```
?       Question mark matches any character in filename patterns.
≈       Approximately (Option-X) matches any string in patterns.
[…]     Brackets enclose character sets in filename patterns.
*       Star indicates zero or more repetitions in patterns.
+       Plus indicates one or more repetitions in patterns.
« »     European quotes (Option-\ and Option-Shift-\) enclose
        repeat counts.
```

Do not be concerned if you do not understand the meaning of each special character now. The special characters you need to understand are discussed throughout this guide and are summarized in the appendix "Table of Special Characters and Operators."

Some of the characters, such as the question mark, are typed with familiar keys on the keyboard. Others, such as the ≈ character (Option-X), are typed with an Option-key combination.

You do not need to memorize dozens of combination keystrokes that create special characters. To look up the keystroke for any of the characters typed with an Option-key combination, you use the Key Caps desk accessory that comes with the Macintosh system software. After you open Key Caps, press the Option key to view the optional characters (Figure 1-15).

Figure 1-15 The Key Caps desk accessory, with the Option key pressed



In Figure 1-15, you can see several of the characters you have already read about in this chapter, such as the • and ≈ characters. To type the • character, you would press Option-8; and to type the ≈ character, you would press Option-X. For more information about using Key Caps, refer to your Macintosh guide to System 7.

For more practice with using the `Help` command and the special characters, be sure to complete the steps in the following section, "Tutorial—Getting Online Help."

Tutorial—Getting Online Help

This tutorial demonstrates how to use the MPW Shell's online help. In this tutorial, you will

- use the `Help` command
- enter commands that contain special characters
- display a list of shortcuts you can use as you work with the MPW Shell

To practice using online help in the MPW Shell, follow these steps:

1. **Place the insertion point at the end of the help text in the Worksheet window.**

2. **Enter the command**

   ```
   Help FileSystem
   ```

   The MPW Shell lists the commands you can use to manage files and directories. The first few lines in the list are:

   ```
   Help Catenate           # concatenate files
   Help Close              # close specified windows
   Help Compare            # compare text files
   ```

   Notice the line that describes the `Files` command:

   ```
   Help Files       # list files and directories
   ```

3. **Enter the command**

   ```
   Help Files
   ```

   The MPW Shell displays the complete syntax of the `Files` command. The first few lines of the syntax look like this:

   ```
   Files            # list files and directories
   Files [option…] [name…] > fileList
       -c creator     # list only files with this creator
       -d             # list only directories
   ```

   You can enter the `Files` command with any of its options to generate lists of files that contain the information you specify.

4. **Now enter the command**

   ```
   Help Characters
   ```

   The MPW Shell displays a list of some of the special characters you can use with MPW Shell commands, similar to the list you saw in the last section.

   Notice these two lines in the list:

   ```
   ?       Question mark matches any character in filename patterns.
   ≈       Approximately (Option-X) matches any string in patterns.
   ```

   These lines tell you that you can use the ? and ≈ characters with the `Files` command to match filename patterns.

   The ? character matches any single character, and the ≈ character (Option-X) matches any string of characters.

5. **Enter this command, using Option-X to type the ≈ character:**

   ```
   Files ≈
   ```

The MPW Shell displays a list of all the files in the MPW directory, something like this, but longer:

```
:Scripts:
BuildCommands
BuildMenu
BuildProgram
CompareFiles

:Tools:
AboutBox
Asm
Backup
C
Canon
```

The names that begin and end with colons (like `:Scripts:` and `:Tools:`) are directory names; the list beneath each directory name shows the files it contains.

6. **Now enter the command**

```
Help Shortcuts
```

The MPW Shell displays a list of tips you can use to save time.

Most of the shortcuts are described in this guide, but you can leave the list in the Worksheet window as a reference.

Notice the third line in the list:

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

7. **Scroll back to the list of special characters that you created in step 4.**
8. **Place the insertion point on the first / character in the line**

```
/…/           Slash quotes all characters except ∂, {, and "
```

9. **Double-click.**

All of the text between the two slash marks is selected.

In MPW Shell commands, you often see text within a pair of delimiters, such as slash marks, parentheses, brackets, braces, or quotation marks. Double-clicking on either character in the pair is a convenient way to select all of the text within the delimiters.

10. Either leave the text you have added in the Worksheet window for future reference or delete it by selecting it and pressing Delete.

As you have learned from this tutorial, the online help system is always available as you work with the MPW Shell. Remember that to get help on the MPW Shell, you enter the `Help` command. To get technical information about programming the Macintosh, use the Macintosh Programmer's Toolbox Assistant.

Building a Sample Application

MPW comes with a variety of compilers and assemblers that produce code for different runtime environments. These tools can vary from release to release but generally include C, C++, and assembler programs for the Macintosh 680x0, Power Macintosh, and CFM-68K runtime environments. Each of the compilers and assemblers comes with sample applications that are stored in the Examples directory.

In general, to build any of the MPW sample applications, you first need to create a **makefile**, a text file that contains all of the commands needed to build the application. The easiest way to create a makefile is by using the Build menu. (However, when you build your own large applications, you will probably write your own makefiles. Writing makefiles is described in detail in *Building and Managing Programs in MPW*.)

You can practice building a sample application by working through the steps in the next section, "Tutorial—Building and Running SillyBalls."

Tutorial—Building and Running SillyBalls

This tutorial is for developers who use the C programming language. If you do not have a C compiler installed, you will not be able to complete this tutorial. If you do, you can build and run SillyBalls using any Macintosh computer that meets MPW's system requirements and any monitor that you can connect to your Macintosh. However, you will get best results on a color monitor.

In this tutorial, you will build the C sample application named SillyBalls. In order to build SillyBalls, you will

- determine the directory in which the files will be stored

- use the Build menu to create a makefile
- build the application according to the rules the makefile contains
- run the application
- make a change to the source file
- build and run the application again

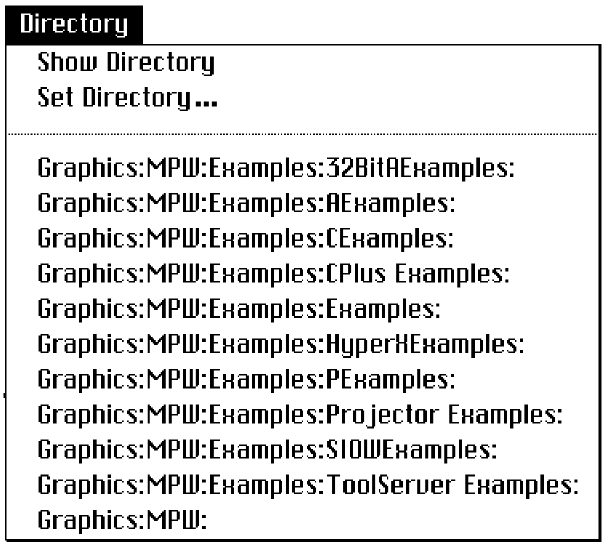
If you open the CExamples directory, you can see that a makefile for SillyBalls, named SillyBalls.make, already exists. This tutorial asks you to rename the existing makefile and build another one so that you can practice building a makefile. After you complete this tutorial, you can continue to build other sample applications by using the makefiles that are already provided in the directories within the Examples directory.

To build and run SillyBalls, follow these steps:

1. **Open the Directory menu. From the bottom part of the menu, choose the long directory name that ends with CExamples.**

   As shown in Figure 1-16, the bottom part of the Directory menu lists the full names of some of the directories on your hard disk.

Figure 1-16 Choosing CExamples from the Directory menu


Directory
Show Directory
Set Directory…
Graphics:MPW:Examples:32BitAExamples:
Graphics:MPW:Examples:AExamples:
Graphics:MPW:Examples:CExamples:
Graphics:MPW:Examples:CPlus Examples:
Graphics:MPW:Examples:Examples:
Graphics:MPW:Examples:HyperXExamples:
Graphics:MPW:Examples:PExamples:
Graphics:MPW:Examples:Projector Examples:
Graphics:MPW:Examples:SIOWExamples:
Graphics:MPW:Examples:ToolServer Examples:
Graphics:MPW:

In Figure 1-16, the CExamples directory is the third item in the list in the bottom part of the menu.

Each directory in the list is shown with its **full pathname,** which begins with a hard disk name, followed by any intermediary directory names, and ending with the final directory name. Each part of the pathname is separated from the next by a colon. If a pathname is too long to fit in the menu, it is truncated and displayed with an ellipsis in the middle. Pathnames are described in more detail in the chapter "Working With Files and Directories."

Choosing CExamples from the Directory menu makes it the **current directory,** so that any new files you create are stored there by default.

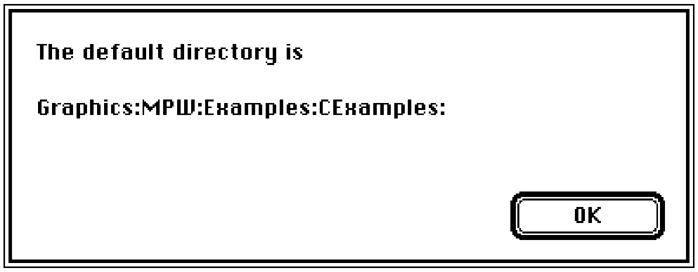
2. **Choose Show Directory from the Directory menu.**

   A dialog box like the one shown in Figure 1-17 appears.

   The dialog box confirms that CExamples is the current directory. The dialog box that you see shows the name of your hard disk, rather than Graphics, at the beginning of the pathname.

   If the dialog box does not show that CExamples is the current directory, complete steps 1 and 2 again.

Figure 1-17 Checking the current directory



```
The default directory is

Graphics:MPW:Examples:CExamples:
```

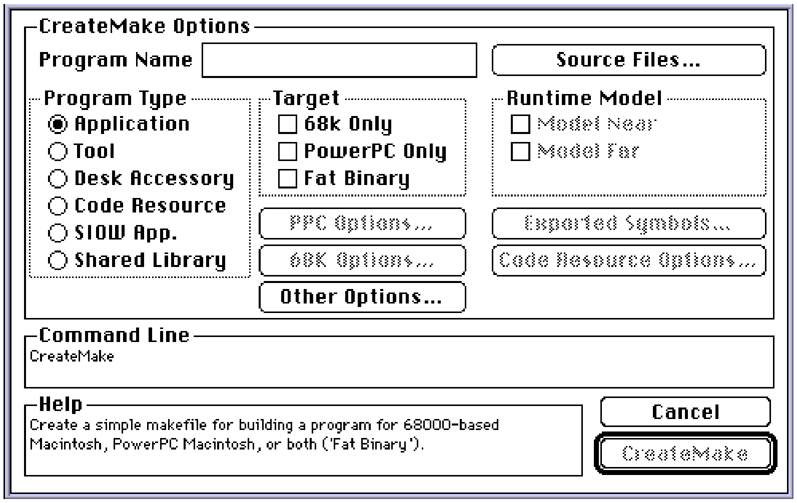
3. **Click OK to dismiss the dialog box.**
4. **From the Finder, double-click the icon for the Examples directory. When the directory opens, double-click the icon for the CExamples directory.**
5. **Rename the file SillyBalls.make to SillyBalls.make.orig (or to another name, if you prefer).**

Even though SillyBalls already has a makefile, you will use the Build menu to create a new makefile so that you can see how the process works.

6. **Choose Create Build Commands from the Build menu.**

   The Commando dialog box for the `CreateMake` command appears (Figure 1-18).

Figure 1-18 Creating a makefile



Program Type: [✓ Application]

CreateMake

Create a simple makefile for building a program for 68000-based Macintosh, PowerPC Macintosh, or both ('Fat Binary').

You can use the `CreateMake` command to create a makefile.

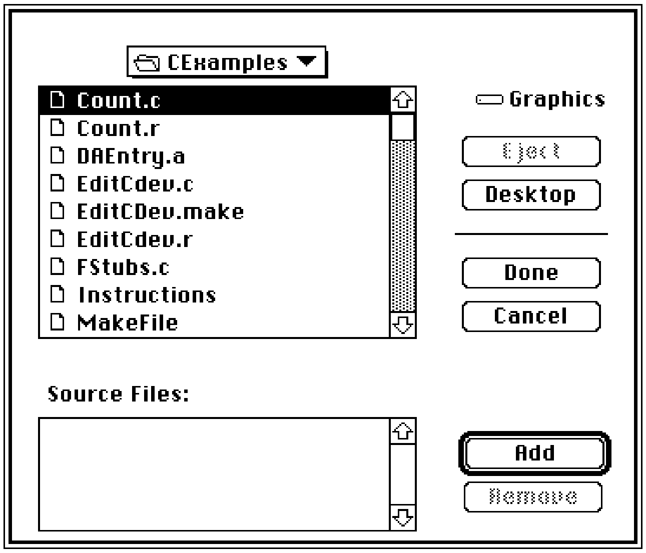
7. **Complete the `CreateMake` Commando dialog box:**

   □ Type SillyBalls in the Program Name box.
   □ Click the Application button.
   □ Click the 68k Only checkbox.

8. **Click the Source Files button.**

   A standard file dialog box showing a list of the files in the CExamples directory appears, as shown in Figure 1-19.

Figure 1-19 Choosing the source files for the build process



9. **Select SillyBalls.c, then click Add.**

   SillyBalls.c is added to the Source Files list at the bottom of the dialog box. To build SillyBalls, you only need to add one source file to the list. However, for other applications, you can add as many as necessary, one at a time.

   The standard file dialog box should now look like the one shown in Figure 1-20.

10. **Click Done.**

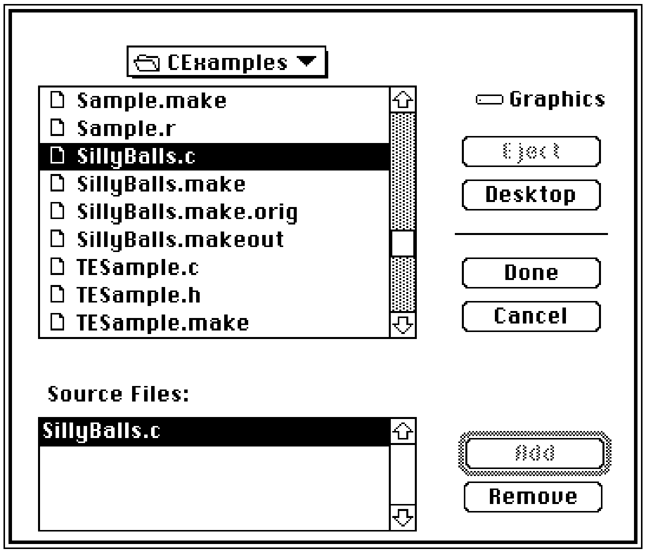
    You return to the `CreateMake` Commando dialog box, shown earlier in Figure 1-18.

11. **Click CreateMake.**

    The `CreateMake` command creates the makefile. As `CreateMake` works, you see the spinning beachball cursor.

    When `CreateMake` is finished, it stores the new makefile, named SillyBalls.make, in the CExamples directory. Now you must build the application according to the rules in the makefile.

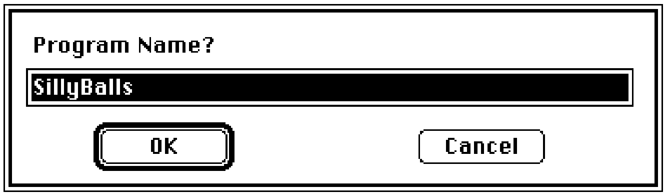
Figure 1-20 The standard file dialog box, once the source files are added



12. Choose Build from the Build menu.

The MPW Shell presents a dialog box (Figure 1-21) displaying the application name you entered in step 7.

Figure 1-21 Confirming the application's name



13. Click OK.

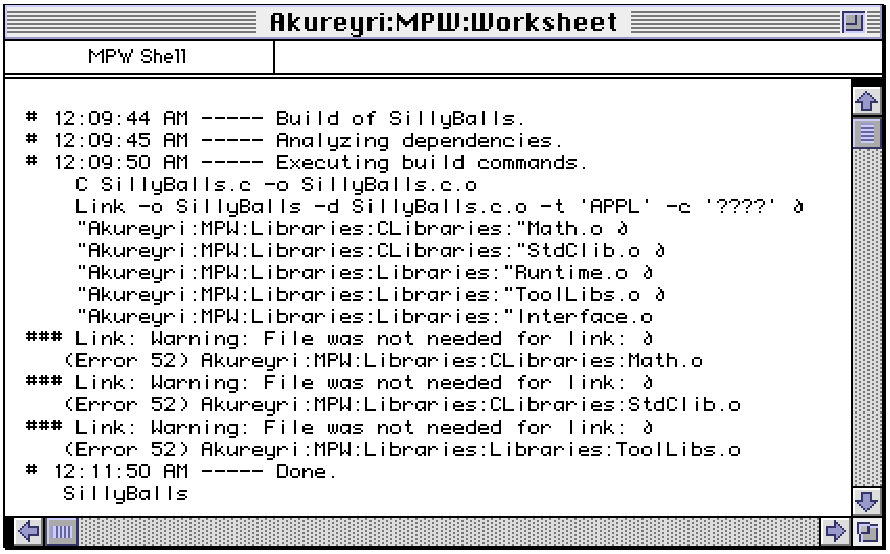
MPW builds SillyBalls from the source file SillyBalls.c, according to the commands in SillyBalls.make.

As MPW builds the program, it creates messages about the build process, and displays them in the Worksheet window (Figure 1-22).

The messages might run past the right edge of your Worksheet window. If so, you can simply make the window larger so that you can read them.

(In Figure 1-22, the messages have been spread across several lines, so that you can read them more easily. In order to spread the commands across several lines, a ∂ character [Option-D] has been added at the end of each line. You won't see the ∂ character in the messages that appear on your screen. You will learn more about the ∂ character in the chapter "Entering Commands.")

Figure 1-22 Watching the build process in the Worksheet window



```
# 12:09:44 AM ----- Build of SillyBalls.
# 12:09:45 AM ----- Analyzing dependencies.
# 12:09:50 AM ----- Executing build commands.
    C SillyBalls.c -o SillyBalls.c.o
    Link -o SillyBalls -d SillyBalls.c.o -t 'APPL' -c '????' ∂
    "Akureyri:MPW:Libraries:CLibraries:"Math.o ∂
    "Akureyri:MPW:Libraries:CLibraries:"StdClib.o ∂
    "Akureyri:MPW:Libraries:Libraries:"Runtime.o ∂
    "Akureyri:MPW:Libraries:Libraries:"ToolLibs.o ∂
    "Akureyri:MPW:Libraries:Libraries:"Interface.o
### Link: Warning: File was not needed for link: ∂
   (Error 52) Akureyri:MPW:Libraries:CLibraries:Math.o
### Link: Warning: File was not needed for link: ∂
   (Error 52) Akureyri:MPW:Libraries:CLibraries:StdClib.o
### Link: Warning: File was not needed for link: ∂
   (Error 52) Akureyri:MPW:Libraries:Libraries:ToolLibs.o
# 12:11:50 AM ----- Done.
   SillyBalls
```

At this point, the build messages probably look a bit complex to you. They include

- a `C` command, which compiles the source file into an object file
- an `ILink` command, which links the object file with library files to create an application

Do not be concerned about the messages that begin with

```
*** Link: Warning:
```

These messages do not imply a serious error. Rather, they explain that some of the library files specified in the `Link` command were not necessary.

When the build process is complete, the object file SillyBalls.c.o and the SillyBalls application (Figure 1-23) are stored in the CExamples directory.

Notice that the insertion point is placed just after the program name SillyBalls in the Worksheet window.

Figure 1-23 The SillyBalls icon

14. **To run SillyBalls, press Enter or double-click the SillyBalls icon in the CExamples directory.**

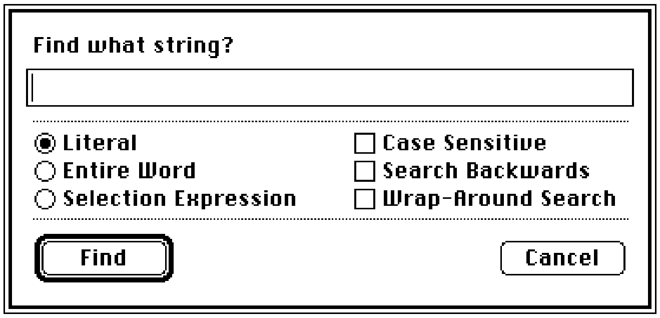
    The application starts and displays randomly colored ovals in a window named Bo3b Land.

    Because your users might think that the window name has an error in it, you can change `Bo3b` to `Bob` in the source file, then rebuild the application.

15. **Open the file named SillyBalls.c, then choose Find from the Find menu.**

    The Find dialog box (Figure 1-24) appears.

Figure 1-24 The Find dialog box



16. **Type** `Bo3b` **in the "Find what string" box, then click Find.**

    The first occurrence of `Bo3b` in the file is selected. Because it is within a comment, you can skip it.

17. **Choose Find Same from the Find menu to locate the next occurrence of** `Bo3b`**. This time, change** `Bo3b` **to** `Bob`**.**

18. **Choose Save from the File menu to save the file.**

19. **Choose Build from the Build menu. When the program name dialog box appears, click OK.**

    The MPW Shell rebuilds SillyBalls, displaying its build messages in the Worksheet window again.

20. **To run SillyBalls again, press Enter or double-click the SillyBalls icon.**

    This time, the application opens a window named Bob Land.

21. **To stop SillyBalls, click in the Bob Land window.**

You have now created a makefile and built a sample application. At this point, you may want to experiment with some of the other MPW sample applications. To build them, you can use the steps outlined in this tutorial. Remember that with the other sample applications, you do not need to build a makefile first because a makefile is already provided.

C H A P T E R 2

Using Commando Dialog Boxes

Contents

One of the most useful features of the MPW Shell is that you can execute any command or tool by clicking on controls in a dialog box, rather than by typing a command line in a window. You can use the dialog boxes, known as Commando dialog boxes, to compose a syntactically perfect command, even when you don't know the command's syntax. Even experienced developers use Commando dialog boxes, when they write complex commands or commands with unfamiliar syntax.

Of course, you can always type a command line, if you prefer. Quite often, it is easier to execute commands from Commando dialog boxes, but you are not required to do so. By using a Commando dialog box, you can either execute the command right away or have the MPW Shell add it to a window so that it can be executed later.

This chapter describes how to use Commando dialog boxes, specifically, how to

- use a Commando dialog box to execute a command
- use a Commando dialog box to add a command to a file or window
- change the size or position of controls within the Commando dialog box
- use two specialized Commando dialog boxes, for the `UserVariables` and `Commando` commands

If you are not familiar with Commando dialog boxes or if you are new to the MPW Shell, read this chapter and complete its tutorials; otherwise, continue with the chapter "Entering Commands."

Executing a Command

To open a Commando dialog box so that it executes a command, which is how you usually use Commando dialog boxes, you type the command name and then press Option-Enter. For example, to open the `Print` Commando dialog box, you can use

```
Print   <Option-Enter>
```

If you prefer, you can press Option-Command-Return at the end of the command, instead of Option-Enter. For example, you can also use

```
Print   <Option-Command-Return>
```

to open the `Print` Commando dialog box.

A third method of opening a Commando dialog box is to type the command name, then press Option-semicolon to type an ellipsis, and then press Enter. For example, to open the `AddMenu` Commando dialog box, you can type

```
AddMenu…   <Enter>
```

An ellipsis looks like three periods, but it isn't. Be sure to type the ellipsis using Option-semicolon. If you type three periods, the command does not work. You can type the ellipsis anywhere on the line following the command. For example, you could also have used either of the commands

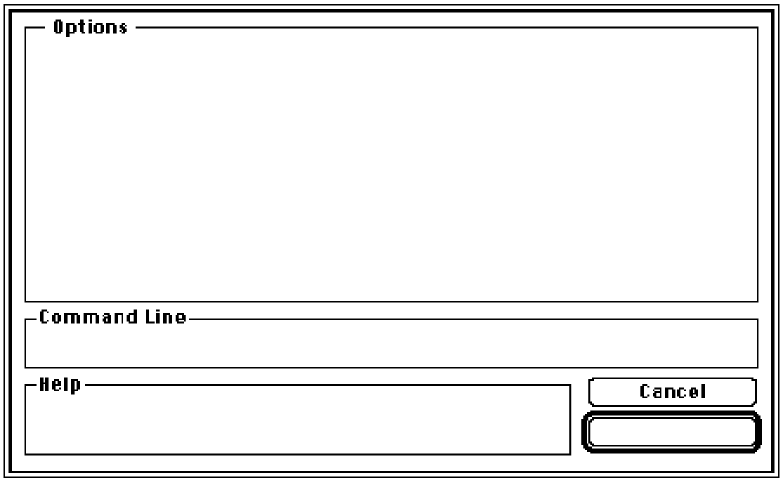
```
AddMenu …   <Enter>
AddMenu        …   <Enter>
```

to open the `AddMenu` Commando dialog box.

The Parts of a Commando Dialog Box

Each Commando dialog box is unique, but they all follow a similar pattern, which is shown in Figure 2-1.

Figure 2-1 The parts of a Commando dialog box



Each Commando dialog box has

- an Options box
- a Command Line box
- a Help box
- a Cancel button
- a command button, in the lower right

The Options box contains checkboxes and radio buttons to select, text boxes to type in, and pop-up menus to choose from. As you complete the items in the Options box, the command in the Command Line box changes. Clicking a radio button, for example, usually adds an option to the command line. If you change your mind and click another button, the option is removed or changed.

The Help box gives you online help with any of the parts of the Commando dialog box. When you first open the Commando dialog box, the Help box displays a description of the command. To get help on any of the parts of the Commando dialog box, place the pointer over the part's label and press the mouse button.

As you compose the command, the command button is dimmed until you have clicked on all of the controls required to execute a command. When you have completed the necessary controls, the command button becomes active so that you can click it.

To execute the command, you click the command button; to cancel the command and dismiss the Commando dialog box, you click the Cancel button. To execute the command *and* write it to the active window, you press the Option key while you click the command button. This method is useful when you want to save the command so that you can execute it again later. You'll learn more about this in the tutorial that follows, "Tutorial—Adding a Menu."

Tutorial—Adding a Menu

In this tutorial, you will use the `AddMenu` Commando dialog box to add a menu to the MPW Shell's menu bar. The menu that you will add, named Shortcuts, contains three items that you will find useful as you work with the MPW Shell.

Each item in the Shortcuts menu opens a Commando dialog box. The items you will add to the menu are

- Print… , which opens the `Print` Commando dialog box
- Quote… , which opens the `Quote` Commando dialog box
- Files… , which opens the `Files` Commando dialog box

You have probably already seen the … character in the menus of other Macintosh applications. In general, the … character means that the menu item opens a dialog box. In the menu that you will create in this tutorial, the … character shows that the menu item opens a specific type of dialog box, a Commando dialog box.

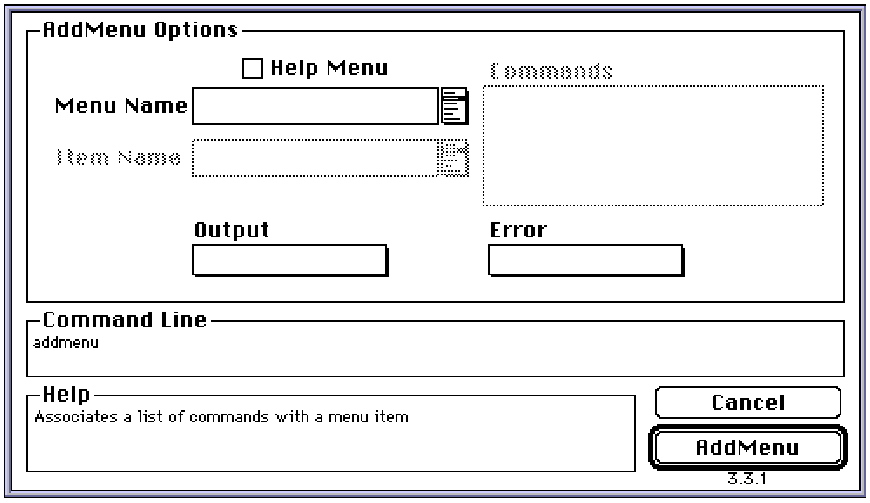
Note
If you make an error while completing this tutorial, you can click the Cancel button in the Commando dialog box, then begin again. ◆

To add a menu using the `AddMenu` Commando dialog box, follow these steps:

1. **Type `AddMenu` in the Worksheet window, then press Option-Enter.**

   The `AddMenu` Commando dialog box appears, shown in Figure 2-2 on page 2-6.

Figure 2-2 The AddMenu Commando dialog box



Command Line
addmenu

Help
Associates a list of commands with a menu item

The Command Line box displays the command

```
AddMenu
```

without any options or parameters following it.

2. **Type `Shortcuts` in the Menu Name box.**

   The Menu Name box contains the title of the menu that will appear in the menu bar.

   After you add the menu title, the command in the Command Line box changes to

   ```
   AddMenu Shortcuts
   ```

3. **Type `Print…` in the Item Name box, using Option-semicolon to type the … character.**

   The Item Name box shows how the item will appear in the menu.

   The command in the Command Line box changes to

   ```
   AddMenu Shortcuts 'Print…'
   ```

The single quotation marks around `Print…` show that the word and the character following it are one parameter. You will learn more about using quotation marks in commands in the chapter "Entering Commands."

4. **Type `Print…` in the Commands box, using Option-semicolon to type the … character.**

   The Commands box contains the commands that are executed when the item is chosen from the menu.

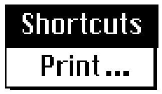
   The Command Line box now displays the complete command, like this:

   ```
   AddMenu Shortcuts 'Print…' 'Print…'
   ```

5. **Hold down the Option key, and click AddMenu.**

   The MPW Shell executes the command, adding the Shortcuts menu to the menu bar, and displays the command in the Worksheet window. At this point, the Shortcuts menu contains only one item, Print... , as shown in Figure 2-3.

Figure 2-3 The Shortcuts menu with one item



In the next few steps, you will add two more items to the Shortcuts menu.

6. **Open the `AddMenu` Commando dialog box again.**
7. **Now create another `AddMenu` command:**
   - □ Type `Shortcuts` in the Menu Name box.
   - □ Type `Quote…` in the Item Name box.
   - □ Type `Quote…` in the Commands box.
8. **Hold down the Option key, and click AddMenu.**

   A second item, Quote… , is added to the Shortcuts menu, and its corresponding `AddMenu` command is displayed in the Worksheet window.

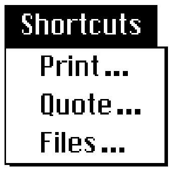
   The `Quote` Commando dialog box is useful for adding expressions and filenames to MPW Shell commands. You will learn more about the `Quote` Commando dialog box in the chapter "Entering Commands."

9. **Open the `AddMenu` Commando dialog box a third time.**
10. **Now build a third `AddMenu` command:**
    - □ Type `Shortcuts` in the Menu Name box.
    - □ Type `Files…` in the Item Name box.
    - □ Type `Files…` in the Commands box.
11. **Hold down the Option key, and click AddMenu.**

    A third item, Files…, is added to the Shortcuts menu and its corresponding `AddMenu` command is displayed in the Worksheet window.

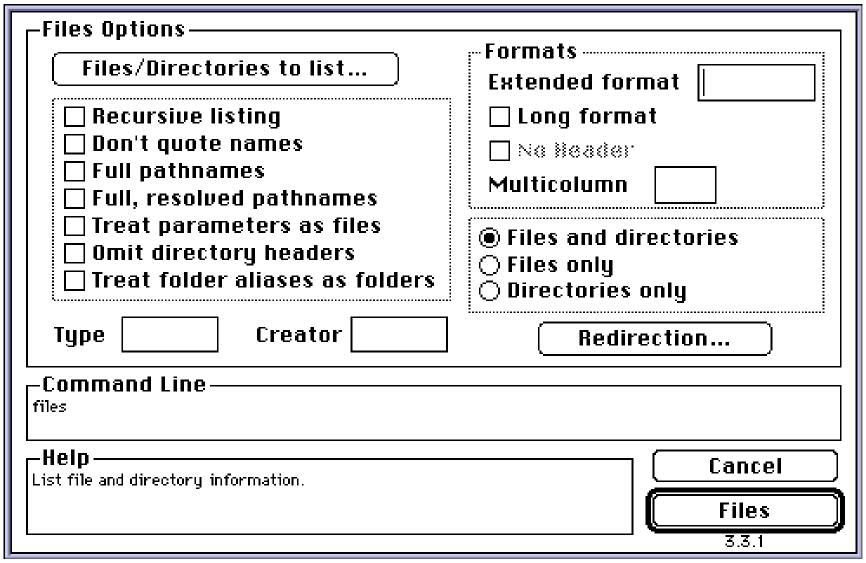
    At this point, the Shortcuts menu should look like the one shown in Figure 2-4.

Figure 2-4 The complete Shortcuts menu



You can now use the Shortcuts menu just like any other MPW Shell menu. For example, when you choose Files, the Commando dialog box for the `Files` command appears (Figure 2-5). You can use it to generate a list of files for any directory or directories on your hard disk.

Figure 2-5 The Files Commando dialog box



Files Options

Files/Directories to list...

☐ Recursive listing
☐ Don't quote names
☐ Full pathnames
☐ Full, resolved pathnames
☐ Treat parameters as files
☐ Omit directory headers
☐ Treat folder aliases as folders

Type

Creator

Formats

Extended format

☐ Long format
☐ No Header

Multicolumn

● Files and directories
○ Files only
○ Directories only

Redirection...

Command Line

```
files
```

Help

List file and directory information.

Cancel

Files

3.3.1

The menu you have added exists only until you leave the MPW Shell. If you want the menu to appear each time you use the MPW Shell, you need to add the commands that build the menu (that is, the three `AddMenu` commands that are displayed in the Worksheet window) to your UserStartup•*name* file. You will find a sample UserStartup•*name* file in the appendixes.

Displaying a Command Without Executing It

You can also open a Commando dialog box so that it displays a command in a window or adds it to a file, without executing the command. This method of opening a Commando dialog box is useful when you want to write a command or a series of commands to execute later. The command you build is added to the active window, unless you use the Commando dialog box to specify another window or file.

To open a Commando dialog box so that it prints a command without executing it, you can

- open and use the Commando dialog box as already described, then press the Command and Option keys while you click the command button
- open the Commando dialog box by entering the command `Commando` followed by a command name, as in

  ```
  Commando Print
  ```

  When you open a Commando dialog box this way and then click the Command button, the command is added to a window or file.

You can even use the `Commando` command to help you write scripts, which are files of commands that can be executed like small programs. To do so, you would build a series of commands, adding each to a file. The process of combining MPW Shell commands into scripts is described in more detail in the chapter "Writing Scripts" later in this guide.

One Commando dialog box is an exception to the rule. The `UserVariables` Commando dialog box, which allows you to customize your MPW environment, is different in that

- you can open it by typing `UserVariables`, then pressing Enter, instead of Option-Enter
- when you open it by pressing Enter, it displays a command without executing it

You will find an example of how to use the `UserVariables` Commando dialog box in the next section, "Tutorial—Setting Window Variables."

Tutorial—Setting Window Variables

In this tutorial, you will use the `UserVariables` Commando dialog box to set several values that affect how you use windows. Specifically, you will set

- window stacking and tiling so that the Worksheet window is always included in stacked and tiled windows
- the size and location of new windows that you open, to make them more convenient to use

The commands that you build will be added to the Worksheet window. To execute the commands, you will select them, then press Enter.

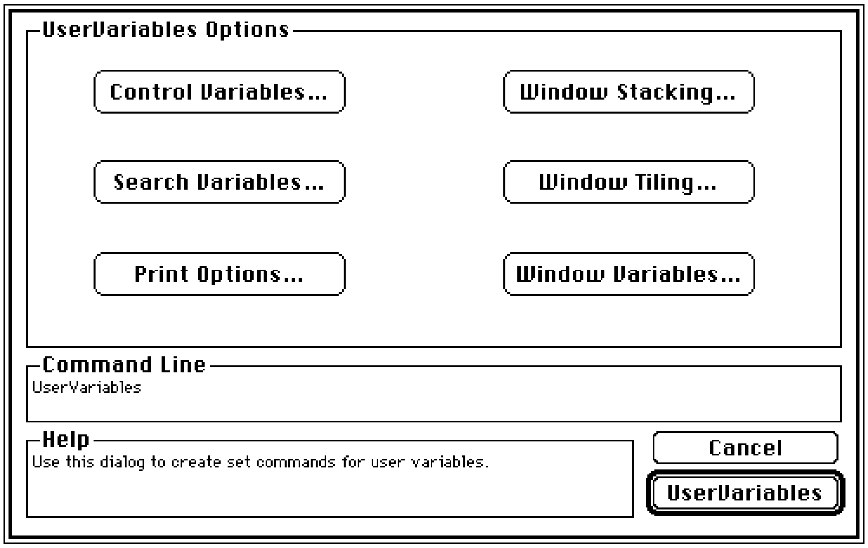
To use the `UserVariables` Commando dialog box, follow these steps:

1. **Type this command in the Worksheet window, then press Enter:**

   ```
   UserVariables
   ```

   The `UserVariables` Commando dialog box appears (Figure 2-6 on page 2-10).

Figure 2-6 The UserVariables Commando dialog box



```
UserVariables
```

Use this dialog to create set commands for user variables.

   Remember that `UserVariables` is an exception to the rule about opening Commando dialog boxes; you can open its Commando dialog box by pressing Enter, instead of Option-Enter.

   You can click on any of the buttons in the `UserVariables` Commando dialog box to open a second dialog box that has controls you can use to change the values you are interested in.

2. **Click the Window Stacking button.**

   The Window Stacking dialog box appears (Figure 2-7), and in it you can change the variables that affect how windows are stacked.

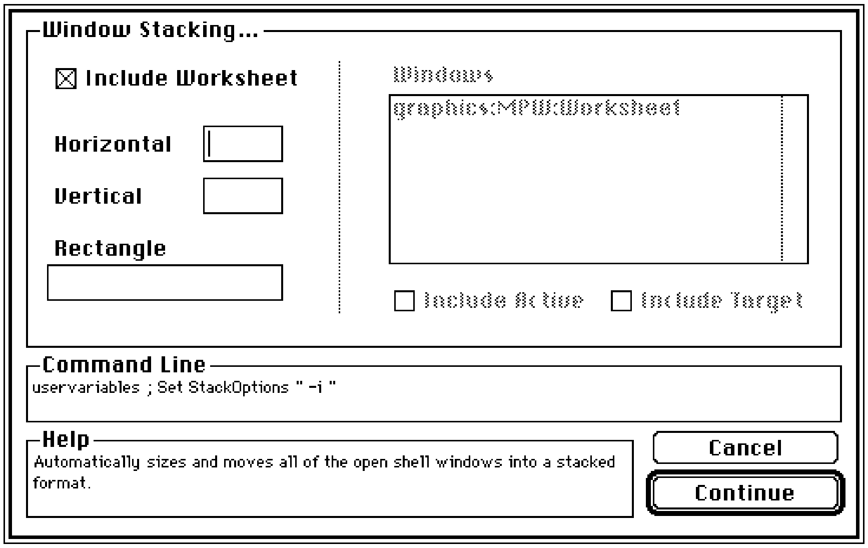
3. **Click Include Worksheet, then click Continue.**

You have just specified that you want the Worksheet window included whenever you stack windows by choosing Stack Windows from the Windows menu. When you click Continue, you return to the `UserVariables` Commando dialog box.

4. **Click Window Tiling.**

The Window Tiling dialog box appears (Figure 2-8).

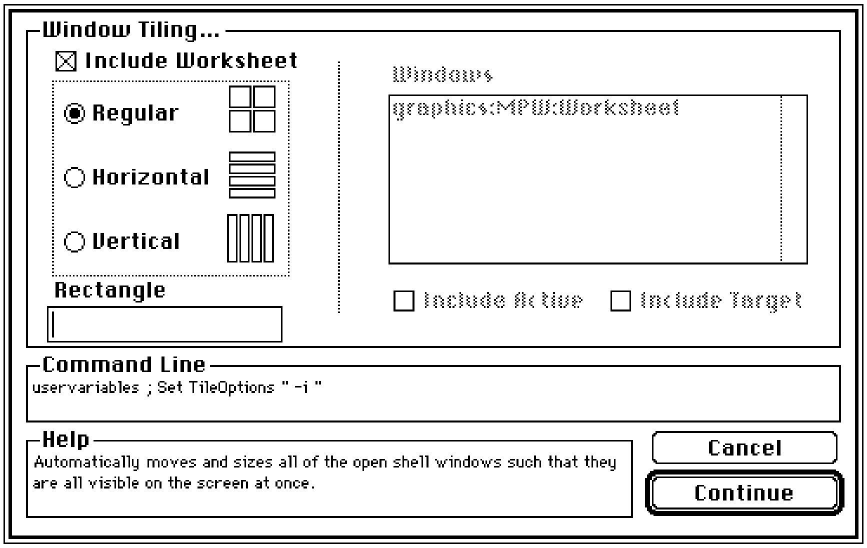
Figure 2-7 The Window Stacking dialog box



```
uservariables ; Set StackOptions " -i "
```

Automatically sizes and moves all of the open shell windows into a stacked format.

Figure 2-8 The Window Tiling dialog box



userVariables ; Set TileOptions " -i "

Automatically moves and sizes all of the open shell windows such that they are all visible on the screen at once.

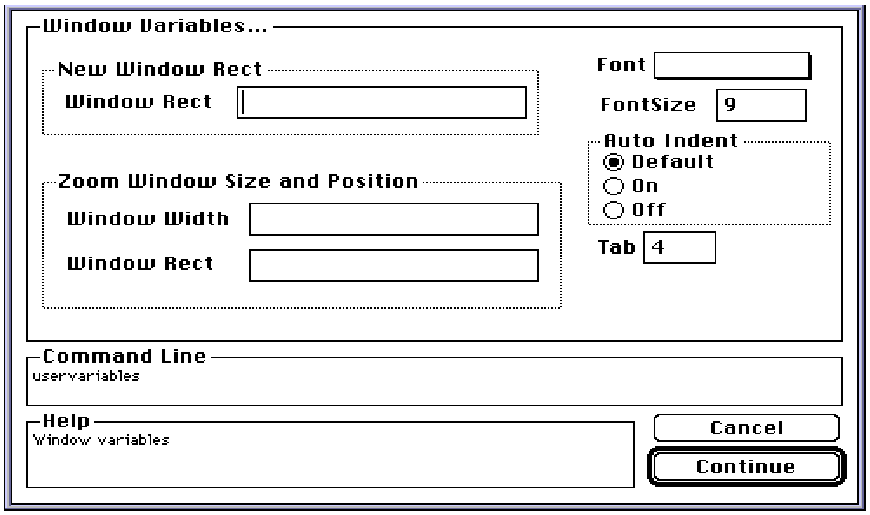
5. **Click Include Worksheet, then click Continue.**

   You have also specified that you want the Worksheet window included when you tile windows by choosing Tile Windows from the Windows menu. When you click Continue, you return to the `UserVariables` Commando dialog box.

6. **Click Window Variables.**

   The Window Variables dialog box appears (Figure 2-9).

Figure 2-9 The Window Variables dialog box



FontSize 9

Auto Indent: [✓ Default] On Off

Tab 4

Command Line: uservariables

Help: Window variables

7. **Enter these coordinates in the New Window Rect box:**

   ```
   10,10,150,300
   ```

   The coordinates `10,10` specify the top-left corner of the window, and the coordinates `150,300` specify the bottom-right corner of the window. The coordinates are measured in pixels and start from the top-left corner of the screen.

8. **Click Continue.**

   You return to the `UserVariables` Commando dialog box.

9. **Click UserVariables.**

   The `UserVariables` dialog box disappears, and the set of commands you have built, beginning with `UserVariables`, is displayed in the Worksheet window.

   The set of commands are a bit long, but the first part of it looks like this:

   ```
   UserVariables ; Set NewWindowRect 10,10,150,300 ; …
   ```

10. **Delete the part of the command that reads**

    ```
    UserVariables ;
    ```

11. **Leave the insertion point in the command, then press Enter.**

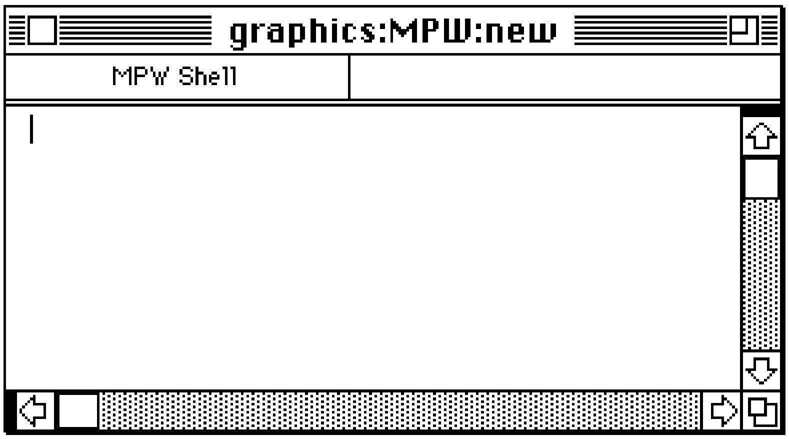
    Now that you have executed the commands, the new values you specified are in effect. If you choose Tile Windows or Stack Windows from the Window menu, the Worksheet window is included; or if you create a new document, it appears in a small window at the top left of your screen.

    The commands you executed remain in the Worksheet window so that you can use them later.

12. **To test the window size you have set, choose New from the File menu.**

    A new, small window opens in the top-left corner of your monitor, just as you specified (Figure 2-10). Note that the new size and location apply to new files you create, but they do not affect windows that are already open.

Figure 2-10 The new window, in its actual size

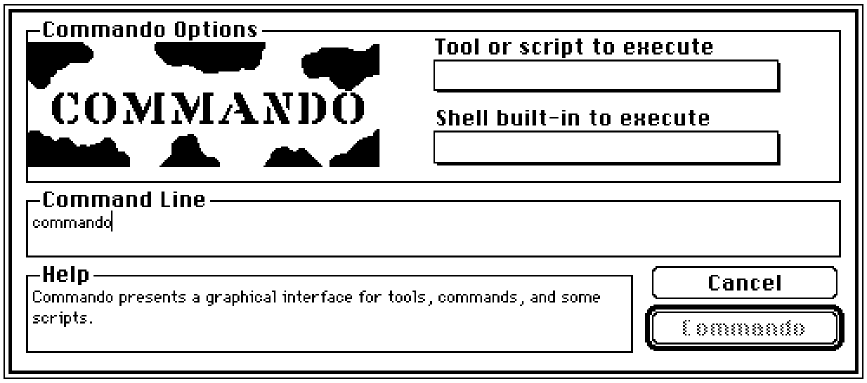


Remember that the changes you have made only take effect until you exit from MPW. If you like the changes and want to set them as defaults so that you can use them each time you use the MPW Shell, you need to add the part of the command line that you executed in step 11 to your UserStartup•*name* file. You will find a sample UserStartup•*name* file in the appendixes.

The Commando Commando Dialog Box

One Commando dialog box deserves special mention: the Commando dialog box that is invoked by the `Commando` command (Figure 2-11).

Figure 2-11 The Commando Commando dialog box



commando

Commando presents a graphical interface for tools, commands, and some scripts.

From the `Commando` Commando dialog box, you can execute any of the MPW Shell's built-in commands, tools, or scripts. From the pop-up menu named "Tool or script to execute," you can choose the item "Select a tool or script to execute." When you do, you are presented with a standard file dialog box from which you can choose any MPW Shell tool or script.

When you open the pop-up menu named "Shell built-in to execute," you see a list of all of the MPW Shell built-in commands. When you choose a command from the list, and then click Commando, the Commando dialog box for that command appears.

If you write your own tools, you can build Commando dialog boxes for them to make them easier to use. For more information on building Commando dialog boxes, see *Building and Managing Programs in MPW*.

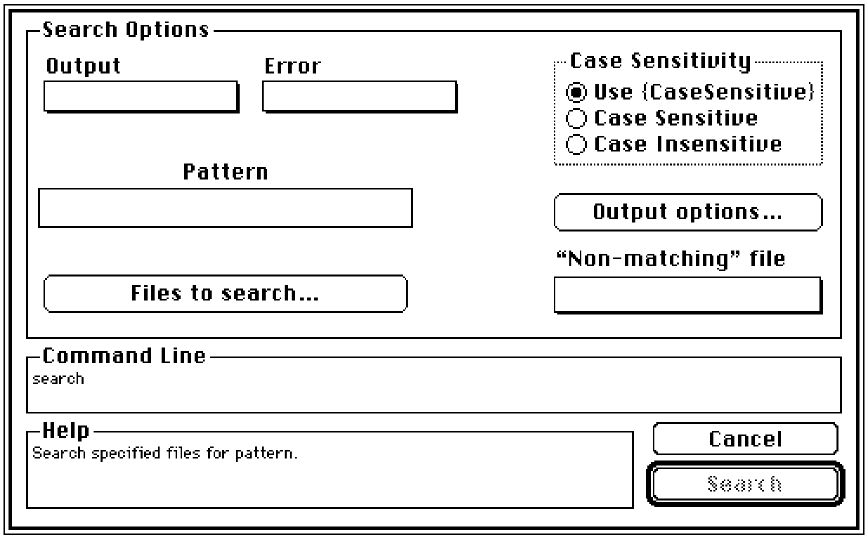
Modifying a Commando Dialog Box

Commando dialog boxes are easy to use, but if there are some that you use often, you may want to move or resize certain elements within them to make them even easier to use. To modify a Commando dialog box, use the `Commando` command with the `-modify` option. For example, the command

```
Commando -modify Search
```

would open the `Search` Commando dialog box (Figure 2-12) so that you can modify it.

Figure 2-12 The original Search Commando dialog box



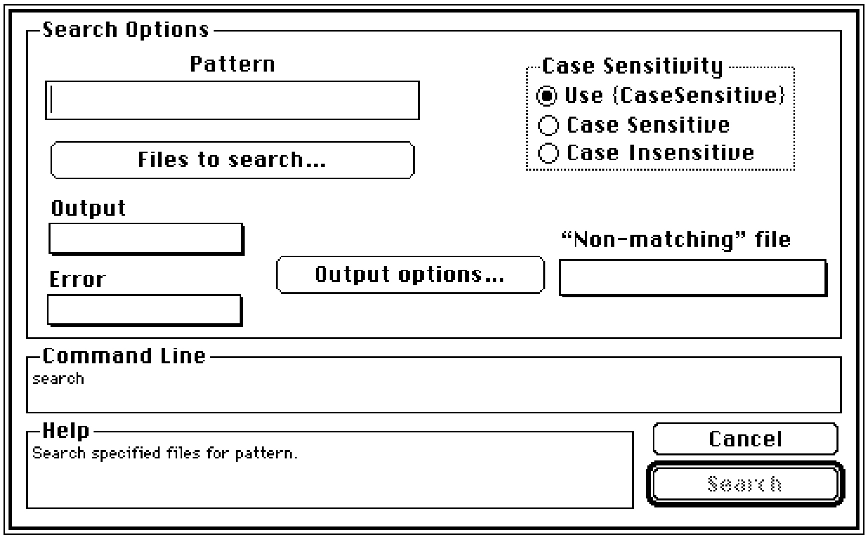
Command Line: search

Help: Search specified files for pattern.

Case Sensitivity: [✓ Use {CaseSensitive}] Case Sensitive Case Insensitive

You might want to modify the `Search` Commando dialog box to make it a bit easier to use, for example, so that it looks like the one in Figure 2-13.

Figure 2-13 A modified Search Commando dialog box



Case Sensitivity: [✓ Use {CaseSensitive}] Case Sensitive, Case Insensitive

Command Line: search

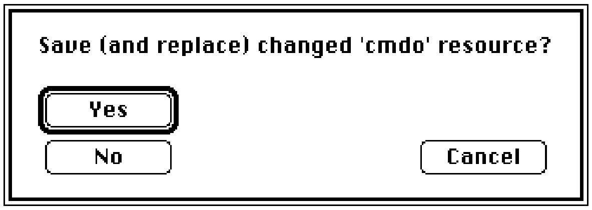
Help: Search specified files for pattern.

To change the size or position of a text field, button, or menu title in the Commando dialog box, hold down the Option key while you click on the control. When you select a control in this way, a small gray rectangle appears in the lower right of the control. When the small rectangle appears, you can resize the control by moving the lower-right corner or you can move the control to another place.

At times you may want to group several items, for example, a text box and its label or a box containing several items. To group several items, press the Option key and click on the first item; then, press both the Option and Shift keys and click on the second item. Once the items are grouped, you can drag them all at once with the mouse.

When you have finished moving and resizing the controls, you click the Cancel button, as if you want to dismiss the Commando dialog box. However, when you click Cancel, the Commando dialog box does not disappear. Instead, a new dialog box appears, asking you to confirm your changes (Figure 2-14).

Figure 2-14 Saving changes to a Commando dialog box


Save (and replace) changed 'cmdo' resource?

You can click Yes to save your changes, No to discard them, or Cancel to continue modifying the Commando dialog box. If you click Yes, the changes are saved so that they appear each time you open the Commando dialog box.

Entering Commands

Contents

Much of the power of the MPW Shell comes from its command language. You can enter any of the MPW Shell's commands interactively in a window, or you can combine commands into scripts. With more than 120 commands, the MPW Shell has its own programming language.

Every developer who uses the MPW Shell should learn its command language. Even though you can execute any MPW Shell command from its Commando dialog box, sometimes the fastest and most powerful way to enter commands is by typing a command line in a window.

Because this chapter is a prerequisite to all of the remaining chapters in this guide, be sure to read all of it. Specifically, this chapter explains how to

- enter simple commands
- use a command as a parameter to another command
- give any command a new name
- define and use variables in commands
- use quotation marks properly to mark certain parameters
- redirect the input and output of commands

Once you complete this chapter, you can move on to any of the remaining chapters in this guide.

Some Basic Command Skills

The simplest way to execute a command is to type a command name, then press Enter. For example, if you enter

```
Beep
```

your Macintosh beeps, producing whatever sound is set in the Sound control panel. If you enter

```
Date
```

the MPW Shell displays the date in the active window, like this:

```
Tuesday, April 6, 1993 10:01:39 AM
```

A command can also take options and parameters. **Options** always begin with a hyphen (for example, `-b` or `-v`) and specify *how* a command is to be carried out. **Parameters** can take many forms and specify *what* the command acts upon.

For example, you can add the parameter `2E,40` to the Beep command, as in

```
Beep 2E,40
```

to make your Macintosh sound a high E note, two octaves above the middle octave. You can also add the `-t` option to the `Date` command, as in

```
Date -t
```

which displays just the time, like this:

```
10:04:01 AM
```

The basic structure of an MPW Shell command is

CommandName [*options…*] [*parameters…*] *CommandTerminator*

In general, you can place options before parameters or parameters before options; or you can mix options and parameters. However, some commands take their parameters in a certain order, and some options themselves take parameters. You may want to refer to the *MPW Command Reference* or to a command's Commando dialog box if you are unsure of the syntax of a specific command.

Every command you type ends with a character called a **command terminator.** The most common command terminator is the Enter character, which sends a command to the MPW Shell to be executed.

By entering commands with various MPW Shell special characters, you can spread commands across several lines, type several commands on one line, or execute commands conditionally, as explained in the following sections.

Continuing a Command to the Next Line

If you type part of a command on a line and then type the ∂ character (Option-D) and then press the Return key, the command continues to the next line. For example, the lines

```
Echo This is really one ∂
 line, not two.
```

form one command. To execute the command, you would select both lines and press Enter. When you do, the MPW Shell displays

```
This is really one line, not two.
```

When you use the ∂ character to end a line, be sure to press Return (not Enter) immediately after the ∂ character. Do not add any spaces or tabs between the ∂ character and the return character.

Note
The ∂-Return character combination that continues the command to the next line is not the command terminator. Instead, the command terminator is the Enter keystroke that executes the command. ◆

If a command has a comment, the comment is not continued to the next line. A comment always ends at the end of the physical line. For example, in the command

```
Echo This sentence makes                        # a comment ∂
 a very, very long command line.    # another comment
```

each line has its own comment. When you enter the command, the MPW Shell displays

```
This sentence makes a very, very long command line.
```

When you use the ∂ character at the end of a line, be sure to follow it with another line. If you end a command line with ∂ without entering another line, the MPW Shell does not execute the command. The ∂ at the end of the line creates a pause and causes the MPW Shell to wait for more input.

In this section, you have seen how to use the ∂ character (Option-D) with a return character to continue a command to the next line. In general, the ∂ character changes the meaning of the following character. You can use the ∂ character in several other ways, as described in the section "Use the ∂ Character to Insert Nonprinting Characters" later in this chapter.

Joining Commands on the Same Line

Sometimes it is convenient to type more than one command on a line, especially when the commands are short and related. To join commands on a line, place a ; character at the end of each command except the last. For example, you can write the three command lines

```
Date
Echo Thank you
Beep
```

as

```
Date; Echo Thank you; Beep
```

You place the ; character after the `Date` and `Echo` commands to show that another command follows on the same line. However, you do not need to place a ; character after the `Beep` command, because it is the last command on the line. After the `Beep` command, you would press Enter to execute the entire command line.

Executing Commands Conditionally

The MPW Shell has two special character combinations, `&&` and `||`, that you can use to execute one command and then execute a second conditionally. If you enter two commands joined by the `&&` character, the second command is executed only if the first command is successful. For example, the line

```
Find /zebra/ && Echo Hooray!
```

searches for the string `zebra` in a text file, and if it finds the string, displays

```
Hooray!
```

in the active window. If you join two commands with the `||` character, the second command is executed only if the first is not successful. For example,

```
Find /elephant/ || Echo Sorry!
```

searches for the string `elephant` in a text file, and if it does not find the string, displays

```
Sorry!
```

in the active window. (If the command finds the string, it selects it.)

Note
You can also use the `&&` and `||` characters in more complex commands, or you can combine MPW Shell commands into **control structures,** which are groups of commands that are executed conditionally or in a certain sequence. Both of these techniques are described in more detail in the chapter “Writing Scripts” later in this guide. ◆

The command termination and continuation characters that have been described in the last few sections are summarized in Table 3-1 and Table 3-2.

Table 3-1 The MPW Shell command terminators

Character	Example	Description
Enter	*cmd* <Enter>	Executes a command
Return	*cmd2cmd1* <Return>	Ends one command and moves *cmd2* to the next line without executing the command
`;`	*cmd1* `;` *cmd2*	Allows you to type more than one command on a single line
`&&`	*cmd1* `&&` *cmd2* <Enter>	Executes the first command and if it succeeds, executes the second
`||`	*cmd1* `||` *cmd2* <Enter>	Executes the first command and if it fails, executes the second

Table 3-2 The MPW Shell command continuation character

Character	Example	Description
∂-Return	*cmd1* … ∂<Return>	Continues a command to the next line … *cmd1*

Using a Command as a Command Parameter

By placing a command within backquotes, you can use its output as a parameter to another command. For example, if you enter

```
Echo The date is `Date`
```

the MPW Shell interprets the `Date` command, passes the output to the `Echo` command, and then displays

```
The date is Tuesday, June 8, 1993 10:26:33 AM
```

Likewise, the command

```
Duplicate `Files SillyBalls.≈` :MPW:
```

generates a list of all the files whose names match the pattern `SillyBalls.≈`, copies the files, and stores the copies in the MPW directory.

A command within backquotes is frequently called an **embedded command,** and the command that uses its output as a parameter is called an **enclosing command.** An embedded command is always executed first, and its output is used as a parameter to the enclosing command.

Renaming MPW Commands

You can give any MPW command a new name, called a **command alias,** with the `Alias` command. Command aliases are useful when you want to make a command name shorter or easier to remember.

Command aliases are not the same as Finder aliases. A command alias is a name you define for an MPW Shell command and is only recognized within the MPW Shell. A Finder alias is an object you create from the Finder that represents another file, directory, or volume. Finder aliases are described in more detail in "Using Finder Aliases in Commands" later in this chapter.

Creating a Command Alias

Suppose that you want to rename the `Files` command (which lists the files and directories in the current directory) to `ls`. You would enter

```
Alias ls Files
```

and the MPW Shell would create the new command name, `ls`. You could then enter `ls` (instead of `Files`) to display a list of the files and directories in the current directory. The original command, `Files`, still works also.

A command alias can take any of the original command's options and parameters. For example, once you have defined `ls` as a command alias for `Files`, you could enter any of these commands:

```
Ls -m 2
Files -m 2
```

If you ever question which MPW Shell command `ls` corresponds to, you can enter

```
Alias ls
```

and the MPW Shell responds with

```
Alias ls Files
```

to tell you that `ls` is a command alias for `Files`. To display a list of all the command aliases you have created, enter `Alias`. The MPW Shell displays a list similar to this one:

```
Alias      File    Target
Alias      Ls      Files
Alias      Ver     Version
```

The command aliases you create by entering `Alias` in the active window take effect only until you restart MPW. To make your command aliases permanent and global, you would add them to your UserStartup•*name* file so that they are read and initialized each time you start MPW. (You can find a sample UserStartup•*name* file in the appendixes.) Once the command aliases are in effect, you can use them on the command line or in a script.

Removing a Command Alias

You can remove a command alias by using the `Unalias` command. For example, to remove the command alias `ls`, you would enter

```
Unalias ls
```

If you enter the command `Unalias` without any command alias names, the MPW Shell removes *all* command aliases that are in effect. However, if you have command aliases defined in your UserStartup•*name* file, nothing is lost. To reinstate the aliases, you can restart MPW, or you can enter a command like

```
Execute UserStartup•mine
```

using the name of your UserStartup•*name* file instead of `UserStartup•mine`.

Sometimes, however, removing all command aliases is just what you want—for example, when you use command aliases in a script and you want to be sure that your users do not have any command aliases defined that conflict with them.

Command Aliases and Commando Dialog Boxes

You can use a command alias to open a Commando dialog box for the original command. To open a Commando dialog box by a command alias, you can

- type the command alias, then press Option-Enter
- type the command alias followed by a … character, then press Enter

For example, if you have defined `ls` as a command alias for `Files`, you could enter either of these commands to open the `Files` Commando dialog box:

```
ls      <Option-Enter>            # opens the Files Commando
ls …                              # also opens the Files Commando
```

However, you cannot use a command alias with the `Commando` command. For example, you could not enter

```
Commando Ls     #this doesn't work
```

but you could enter

```
Commando Files  #this does work
```

because `Files` is the name of an MPW Shell command.

Defining and Using Variables

In MPW Shell commands, you can declare and use as many variables as you like. Variables are useful for

- allowing you to customize your MPW environment
- creating shorter ways of specifying long names
- providing status information to commands and scripts
- defining local values in scripts
- naming the parameters that are passed to a script or a tool

The rules for using variables described in this chapter apply to defining and using variables in MPW Shell commands and scripts, but do not apply to using variables in any of MPW's programming languages. The variables that you use in MPW Shell commands and scripts are often called **Shell variables.**

Defining Your Own Variables

To define variables, you use the `Set` command, followed by a variable name and its initial value. If you enter

```
Set Today Tuesday
```

you have defined a variable named `Today` and set its value to `Tuesday`. Once you define a variable with `Set`, you can use it in any MPW Shell command if you place its name within `{ }` characters, as in `{Today}`.

To check the current value of a variable, you can use `Echo` or `Set`. For example, if you enter

```
Echo {Today}
```

`Echo` responds by printing the value of `{Today}` in the Worksheet window, like this:

```
Tuesday
```

In the `Echo` command, be sure to place the variable name within `{ }` characters. Otherwise, `Echo` displays exactly what you entered. If you enter

```
Echo Today
```

the MPW Shell responds with the exact string you entered, like this:

```
Today
```

You can also use `Set` to check the value of `{Today}`, as in

```
Set Today
```

The MPW Shell responds with the name of the variable and its value, like this:

```
Set Today Tuesday
```

When you define a variable by entering the `Set` command interactively—that is, in the Worksheet window—you can use the variable only with other commands you enter interactively, until you leave MPW. To extend a variable's scope so that you can use it in scripts, you need to use the `Export` command, as described in the chapter "Writing Scripts."

Removing Variable Definitions

You can remove a variable definition with the `Unset` command. For example, if you have already defined the variable `{Today}` to have the value `Tuesday`, the command

```
Unset Today
```

would remove the variable's definition. When you use the `Unset` command, just as when you use `Set`, you enter the variable name without the `{ }` characters. After you remove the variable's definition, if you enter

```
Echo {Today}
```

the MPW Shell displays only a return character (that is, the insertion point moves to the next line) because `{Today}` is no longer defined.

IMPORTANT

When you use `Unset`, be careful to specify a variable name after it. If you enter `Unset` without any parameters, the MPW Shell deletes nearly all of its variable definitions, which could be a problem. Do not enter `Unset` by itself. ▲

The Variables That the MPW Shell Uses

Many variables are already defined as part of the MPW Shell. Some are internal variables that are maintained dynamically, such as the name of the current active window. Other variables are defined in the MPW Shell, but are available for you to redefine. The variables that are part of the MPW Shell, both the internal ones and the ones you can define, are known as Shell variables.

Table 3-3 lists the Shell variables whose values you should not change.

Table 3-3 The MPW Shell variables that are maintained dynamically

Variable name	Initial value and meaning
`{Active}`	*volumeName*:*directoryName…* :*windowName* The full pathname of the current active window
`{Aliases}`	*alias1, alias2, alias3 …* List of all the command aliases that you have defined
`{Boot}`	*volumeName*: The name of the volume where the system software is stored
`{Command}`	*volumeName*:*directoryName…*:*commandName* The full pathname of the last command that was executed
`{MPW}`	*volumeName*:`MPW:` The full pathname of the directory in which the MPW software is stored

Table 3-3 The MPW Shell variables that are maintained dynamically (continued)

Variable name	Initial value and meaning
`{MPWVersion}`	*versionNumber* The version of MPW that you are using
`{PrefsFolder}`	'*volumeName*`:System Folder:Preferences:MPW:'` The directory to search for the UserStartup and Quit files, if they are not located in the directory named in `{MPW}`
`{ShellDirectory}`	*volumeName*`:MPW:` The full pathname of the directory that contains the MPW Shell
`{Status}`	`0` The result of the last command that was executed
`{SystemFolder}`	'*volumeName*`:System Folder:'` The full pathname of the directory that contains the System and Finder files
`{Target}`	*volumeName*:*directoryName*… :*windowName* The full pathname of the target window
`{TempFolder}`	'*volumeName*`:Temporary Items:'` A system folder that contains temporary files that applications create.
`{Windows}`	*volumeName*:*directoryName*… :*windowName*:, … A list of the current windows, with each name separated from the next by a comma
`{Worksheet}`	*volumeName*`:MPW:Worksheet` The full pathname of the Worksheet window

Table 3-4 lists the Shell variables whose values you can change.

Table 3-4 The MPW Shell variables whose values you can change

Variable name	Initial value and meaning
`{AIncludes}`	*volumeName*`:MPW:Interfaces:AIncludes:` The directory that contains the assembly-language include files
`{AutoIndent}`	`1` Whether text in a window is automatically indented
`{CaseSensitive}`	`0` Whether text searches consider case

Table 3-4 The MPW Shell variables whose values you can change (continued)

Variable name	Initial value and meaning
`{CIncludes}`	*volumeName*`:MPW:Interfaces:CIncludes:` The directory that contains the C include files
`{CLibraries}`	*volumeName*`:MPW:Libraries:CLibraries:` The directory that contains the C library files
`{Commando}`	`Commando` The name of the tool that opens dialog boxes that execute commands
`{Commands}`	`:, {MPW}Tools:, {MPW}Scripts:` The list of directories to search to find a command
`{CPlusIncludes}`	*volumeName*`:MPW:Examples:CPLusExamples:CPlusIncludes:` The directory that contains the C++ include files
`{CPlusLibraries}`	*volumeName*`:MPW:Examples:CPlusExamples:CPlusLibraries:` The directory that contains the C++ library files
`{DirectoryPath}`	*volumeName*`:`*directoryName*`,` *volumeName*`:`*directoryName …* A list of the directories you use frequently, to speed changing directories
`{Echo}`	`0` Whether commands are sent to the current location of standard error as they are executed
`{Exit}`	`1` Whether scripts are terminated when an error occurs
`{Font}`	`Monaco` The default text font in new windows and files
`{FontSize}`	`9` The default text size in new windows and files
`{IgnoreCmdPeriod}`	`0` Whether the MPW Shell terminates a command when you type Command-period.
`{Libraries}`	*volumeName*`:MPW:Libraries:Libraries:` The directory that contains the language-independent library files

Table 3-4 The MPW Shell variables whose values you can change (continued)

Variable name	Initial value and meaning
`{MarkAlphabetical}`	`0` If 1, markers are listed alphabetically in the Mark menu. If 0, markers are listed in the Mark menu in order of their occurrence in the text.
`{NewKeyboardLayout}`	0 Whether a new keyboard layout is used
`{NewWindowRect}`	*x1*,*y1*,*x2*,*y2* The size and location of a new window. Contains the distance (in pixels) of the top, left, bottom, and right edges of the window from the corresponding edges of the screen.
`{PInterfaces}`	*volumeName*`:MPW:Interfaces:PInterfaces:` The directory that contains the Pascal interface files
`{PLibraries}`	*volumeName*`:MPW:Libraries:PLibraries:` The directory that contains the Pascal library files
`{PrintOptions}`	`-h` *(print headers)* Options that are used by the Print Window and Print Selection menu items
`{RIncludes}`	*volumeName*`:MPW:Interfaces:RIncludes:` The directory that contains the Rez include files
`{SearchBackward}`	0 Whether a search for a text string moves backward from the insertion point
`{SearchType}`	`0` The type of string an editing command searches for (if 0, a literal string; if 1, a word; if 2, a regular expression)
`{SearchWrap}`	`0` Whether a search for a text string wraps around from the end of a file to the beginning of a file
`{StackOptions}`	`-i | -h … | -v … | -r t,l,b,r` How windows are stacked (if `-i`, includes the Worksheet window; if `-h`, with the horizontal offset specified; if `-v`, with the vertical offset specified; if `-r` followed by top, left, bottom, and right coordinates, within the rectangle the coordinates specify)

Table 3-4 The MPW Shell variables whose values you can change (continued)

Variable name	Initial value and meaning
`{Tab}`	`4` The number of spaces that make up a tab character
`{Test}`	`0` Whether the MPW Shell executes tools and applications when you type their names in a window
`{TileOptions}`	`-i | -h … | -v … | -r t,l,b,r` How windows are tiled (if `-i`, includes the Worksheet window; if `-h`, with the horizontal offset specified; if `-v`, with the vertical offset specified; if `-r` followed by top, left, bottom, and right coordinates, within the rectangle the coordinates specify)
`{TraceFailures}`	`File` *fileName*`;` `Line` *characterNumber* In debugging scripts, the name of the script that failed and the character number (starting from the beginning of the file) where the error occurred
`{User}`	*name* The owner name that is set in the Sharing Setup control panel
`{WordSet}`	a-zA-Z_0-9 The set of characters that constitute a word
`{ZoomWindowRect}`	*x1, y1, x2, y2* The size that a window expands to when you click the window's zoom box. This variable contains the distance (in pixels) of the top, left, bottom, and right edges of the window from the corresponding edges of the screen.

Some of the definitions in Table 3-3 and Table 3-4 use a full pathname. A **full pathname** describes the route one would take to locate a file, and includes a volume name, any necessary directory names, and then the filename. Each part of the pathname is separated from the next by a colon. Pathnames are described in more detail in the chapter "Working With Files and Directories."

You can also enter the command `Set` with no parameters to display the names and current values of all of the variables that are built into the MPW Shell. When you enter `Set`, the MPW Shell displays a list that looks like this one, but is longer:

```
Set ExtendWordSet 1
Set NewKeyboardLayout 0
Set InhibitMarkCopy 1
Set Boot Graphics:
```

In the list, the variable name comes just after `Set`, and the variable's current value comes just after its name.

To redefine the variables listed in Table 3-4, you can use the `UserVariables` Commando dialog box, as described in the section "Tutorial—Specifying Default Formats" on page 4-13.

When to Use Quotation Marks in Commands

When a parameter contains spaces or special characters, you need to place quotation marks around it whenever you use it in a command. This section describes the rules of using quotation marks properly in commands. Because the use of quotation marks can become quite complex, you should read this entire section and understand each of the points presented here.

Quote Parameters That Contain Spaces

In general, when a parameter contains a space, you must place it within quotation marks so that the MPW Shell recognizes it as a single parameter. (The `Echo` command is an exception to this rule.)

For example, you cannot open a file named Chapter 1 with the command

```
Open Chapter 1     #this won't work
```

because the MPW Shell considers `Chapter` and `1` to be two distinct filenames. But you could open it with either of these commands because the filename is enclosed in quotation marks:

```
Open "Chapter 1"
Open 'Chapter 1'
```

The difference between using single and double quotation marks is described in the following sections.

QUOTING RULE #1
Place quotation marks around parameters that contain spaces. ◆

Always Use Quotation Marks in Pairs

Whether you use single or double quotation marks, always use them in pairs. If you enter the command

```
Open "Chapter 1     #this won't work
```

the MPW Shell returns the error message

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

To open the file named Chapter 1, you must place a closing quotation mark after the `1` in the command. This rule seems simple, but it becomes more complex when quotation marks are nested.

QUOTING RULE #2
Always use a single or double quotation mark with a matching single or double quotation mark. ◆

Mark Special Characters Used as Literal Characters

You already know that many characters have special meanings to the MPW Shell. For example, the # character begins a comment, and the ≈ character (Option-X) matches any string of characters. Ordinarily, you use the special characters when you write commands and scripts.

However, you may want to use a special character with its literal meaning. For example, you might enter

```
Echo How are you today?     #this doesn’t work
```

to attempt to display the question

```
How are you today?
```

However, the command does not work, because the ? character is interpreted as a special character. To use a special character as a literal character, you can either enclose it within single quotation marks or place the ∂ character (Option-D) before it. For example, any of these commands would display the sentence you want:

```
Echo How are you today'?'
Echo 'How are you today?'
Echo How are you today∂?
```

because you have cancelled the effect of the ? character.

When you mark a special character with quotation marks, it is best to use single quotation marks, rather than double. Double quotation marks work only for some special characters.

The • character (Option-8) is an exception. Even though it is a special character, you do not need to mark it when you use it in commands. For example, you can open the file named My•File with the command

```
Open My•File
```

QUOTING RULE #3
To use a special character as a literal character, enclose it in single quotation marks or place the ∂ character before it. ◆

Use the ∂ Character to Insert Nonprinting Characters

In general, the ∂ character changes the meaning of the character that follows it. When you use ∂ before the characters `n`, `t`, or `f`, you create a nonprinting character. The combination ∂`n` creates a return character; ∂`t` creates a tab character; and ∂`f` creates a form-feed character. For example, the command

```
Echo How are you ∂n
```

displays

```
How are you
```

in the active window, then adds two Return characters: one that is automatically added by the `Echo` command, and one that is added by the `∂n` character.

The uses of the ∂ character in the MPW Shell are summarized in Table 3-5.

Table 3-5 Uses of the ∂ character

Character combination	Meaning
∂-Return	Continue to the next line; ignore both the ∂ and return characters
∂n	Return character
∂t	Tab character
∂f	Form-feed character
∂-*character*	Ignore this character's special meaning and interpret it as a literal character

QUOTING RULE #4
In a command, `∂n` inserts a return character, `∂t` inserts a tab character, and `∂f` inserts a form-feed character. ◆

Use Single or Double Quotation Marks Appropriately

Single and double quotation marks each have distinct meanings. You use single quotation marks to enclose characters when you want them to be taken literally. Within single quotation marks, special characters are considered literal characters. Single quotation marks are also known as "hard" quotation marks.

You use double quotation marks to enclose variables, special characters used as special characters, command aliases, and embedded commands. When you use double quotation marks, the MPW Shell interprets the characters `{`, `}`, `∂`, and `` ` `` as special characters. In other words, the MPW Shell expands variables, command aliases, embedded commands, and the ∂ character. Double quotation marks are also known as "soft" quotation marks.

When you quote an expression containing one of the special strings `∂n`, `∂t`, or `∂f`, you should use double quotation marks so that the ∂ character is interpreted as a special character and not taken literally.

As a simple example, you can enter a command that displays the value of the variable `{MPW}`, which contains the location of the MPW software. You would enter

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

Note that you place `{MPW}` within double quotation marks because it is a variable that you want expanded. If you placed it within single quotation marks, the `{ }` characters would be taken literally. When you enter the command, it displays the value of the variable `{MPW}`, as in

```
Graphics:MPW:
```

However, the command

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

would display

```
{MPW}
```

in the active window. Table 3-6 lists some examples of expressions within single and double quotation marks and their output.

Table 3-6 Examples of using single and double quotation marks

Command	**Output**
`Echo '∂tHello World'`	`∂tHello World`
`Echo "∂tHello World"`	<tab> `Hello World`
`Echo '∂"Hello World∂"'`	`∂"Hello World∂"`
`Echo "∂"Hello World∂""`	`"Hello World"`
`Echo '{fileName}'`	`{fileName}`
`Echo "{fileName}"`	The filename defined by the variable
``Echo '`Files`'``	`` `Files` ``
``Echo "`Files`"``	The contents of the current directory

QUOTING RULE #5
Use single quotation marks to enclose special characters used as literal characters. Do not use single quotation marks to enclose variables, special characters used as special characters, embedded commands, or command aliases. ◆

QUOTING RULE #6
To enclose variables, special characters used as special characters, embedded commands, and command aliases, use double quotation marks. ◆

Nest Quotation Marks Properly

You can nest a pair of single quotation marks within a pair of double quotation marks, or a pair of double within a pair of single. For example, the command

```
Echo '"Hello World"'
```

has properly nested quotation marks and displays

```
"Hello World"
```

because the single quotation marks cause the MPW Shell to interpret the double quotation marks literally. However, when you nest single within single, or double within double, the MPW Shell interprets the first two quotation marks as one pair and then the second two as another pair, and the command usually does not work. For example, in the command

```
Echo " "Hello World" "
```

the MPW Shell considers the first two quotation marks as one pair enclosing a space, and the second two as another pair enclosing a space. The command would display

```
 Hello World
```

which contains a space, then `Hello World`, then another space. To nest single quotation marks within single quotation marks, or double within double, place the ∂ character before each quotation mark in the inner pair, as shown in Table 3-7.

Table 3-7 Examples of nesting quotation marks

Command	Output
`Echo "'Hello World'"`	`'Hello World'`
`Echo '"Hello World"'`	`"Hello World"`
`Echo " "Hello World" "`	(space) `Hello World` (space)
`Echo " ∂"Hello World∂" "`	(space) `"Hello World"` (space)
`Echo ' 'Hello World' '`	(space) `Hello World` (space)

QUOTING RULE #7
You can nest a pair of single quotation marks within a pair of double quotation marks, or a pair of double within a pair of single. To nest double within double, place the ∂ character before each quotation mark in the inner pair. ◆

Place Variable Names Within Quotation Marks

Whenever you use a variable in a command that finds the value of the variable, you should place the variable name in double quotation marks. You use the quotation marks so that if the variable's definition contains spaces or special characters, they are marked correctly in the command. For example, if you enter

```
Open "{file}"
```

the MPW Shell expands the variable `{file}` and replaces it with a filename. Because the filename might contain spaces or special characters, you place the variable name within quotation marks. For example, if the file is named My List, the command would become

```
Open "My List"
```

which uses quotation marks properly.

An exception to this guideline is a variable that encloses a list of individual items separated by spaces. For example, a variable may contain a list of items through which a command makes a processing loop. If you enclose the variable in double quotation marks, the entire list would be treated as one long string. For a list, you would leave out the double quotation marks so that each item is treated separately. Creating variables that contain lists is described in more detail in the chapter "Writing Scripts."

QUOTING RULE #8
Place variables within double quotation marks, unless they are defined as a list of individual items. ◆

Mark Quotation Marks Used as Literal Characters

To use single or double quotation marks as literal characters, you can

- precede them with the ∂ character
- enclose single quotation marks within double quotation marks, or double within single

Table 3-8 lists some examples.

Table 3-8 Using quotation marks as literal characters

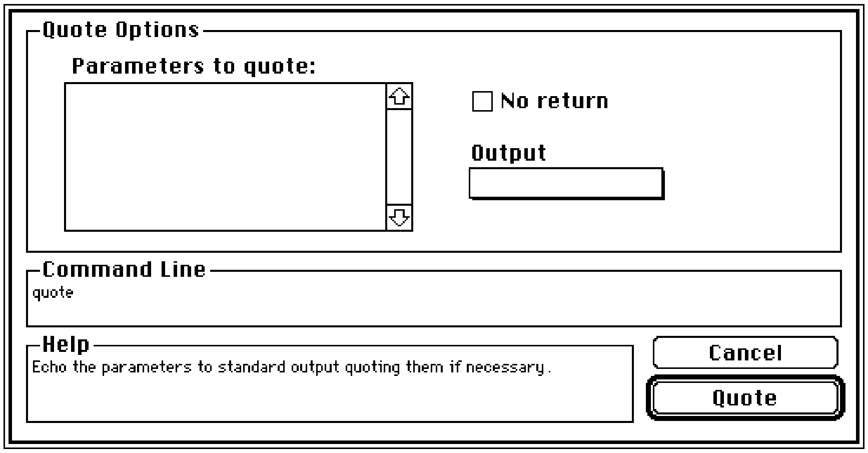
Command	Output
`Echo "hello"`	`hello`
`Echo ∂"hello∂"`	`"hello"`
`Echo '"hello"'`	`"hello"`
`Echo "'hello'"`	`'hello'`

QUOTING RULE #9
Mark quotation marks as literal characters with the ∂ character, or nest them within the opposite type of quotation marks. ◆

Using the Quote Commando Dialog Box

You can let the MPW Shell help you add single quotation marks to an expression by using the `Quote` Commando dialog box (Figure 3-1).

Figure 3-1 The Quote Commando dialog box



quote

Echo the parameters to standard output quoting them if necessary.

Note
If you have completed "Tutorial—Adding a Menu" in the chapter "Using Commando Dialog Boxes," you can open the `Quote` Commando dialog box by choosing `Quote` from the Shortcuts menu. Otherwise, you can open it by typing `Quote`, then pressing Option-Enter. ◆

When you type something in the "Parameters to quote" box, the MPW Shell adds the proper quotation marks to it. Then, when you click the Quote button, the properly quoted parameter is displayed in the active window so that you can use it in a command. For example, if you type

```
Do you think he'll go?
```

then click the Quote button, the MPW Shell displays the expression

```
'Do you think he'∂''ll go?'
```

in the active window. The single quotation mark at `'Do` is closed by `he'`; then `∂'` marks the apostrophe `'` as a literal character; and then another quoted expression is opened at `'ll` and closed at `go?'`.

Because the `Quote` Commando dialog box only adds single quotation marks to expressions, you cannot use it for expressions that contain variables, special characters used as special characters, embedded commands, or command aliases.

When Not to Use Quotation Marks

You should not use quotation marks around a special character used as part of a filename, such as the ≈ character (Option-X) or the ? character. For example, the command

```
Files ≈.c
```

lists all the files in the current directory that end with `.c`. If a filename contains a space, the filename is displayed within single quotation marks.

If you define a variable named `{Dir}` that contains a directory name, you could use the following command to list all the files in that directory that end in `.c`:

```
Files "{Dir}"≈.c
```

Note that the variable name `{Dir}` is placed within quotation marks, but the notation `≈.c` is not. If you enter

```
Files "{Dir}≈.c"
```

you get an error message, unless you happen to have a file named `≈.c`.

Tutorial—Using Quotation Marks in Commands

This tutorial asks you to write some sample commands that use quotation marks, and gives you one or more answers and an explanation at each step. The best way to complete this tutorial is to read a step, write a command for it, and then check your command against the answer and explanation given.

To practice adding quotation marks to commands, follow these steps:

1. **Write a command that opens the file whose name is the current value of the variable `{Filename}`.**

   The command is

   ```
   Open "{Filename}"
   ```

You must place the variable name within double quotation marks because its value might contain spaces or special characters.

2. **Write a command that displays** `Today's date is` **followed by a space and today's date.**

   You can use either of these commands:

   ```
   Echo Today∂'s date is `Date -d`
   Echo "Today's date is `Date -d`"
   ```

   The command `Date -d` generates today's date, with no time. Because the `Date` command is within backquotes, it is an embedded command and its output is passed to the `Echo` command.

   The `Echo` command must display a string that contains an apostrophe. You can mark the apostrophe by preceding it with the ∂ character (Option-D) or by placing it within double quotation marks.

3. **Write a command that displays in the active window**

   ```
   "What's up, doc?" he said.
   ```

   The command is

   ```
   Echo '"What'∂''s up, doc?" he said.'
   ```

   In this command, the string that follows `Echo` is actually made up of three substrings:

   - `'"What'`
   - `∂'`
   - `'s up, doc?" he said.'`

   The first substring has matching single quotation marks that quote the double quotation mark in `"What`. The first substring consists of everything from the beginning of the message up to the apostrophe.

   Next, to properly quote the apostrophe, you must precede it with the ∂ character (Option-D) so that the apostrophe is interpreted as a literal character and not as another single quotation mark.

   Then, you must add another opening single quotation mark, followed by the rest of the string, followed by a closing single quotation mark. Note that the comma, question mark, and double quotation mark in the third substring are all marked by one set of single quotation marks.

4. **Use the** `Alert` **command to display this phrase in an alert box:**

   ```
   " ... and you'll be sorry!"
   ```

To display an alert box that contains a message, you use the `Alert` command followed by the message. The complete command would be

```
Alert '" ... and you'∂''ll be sorry!"'
```

This example is similar to the last one, because it has three substrings:

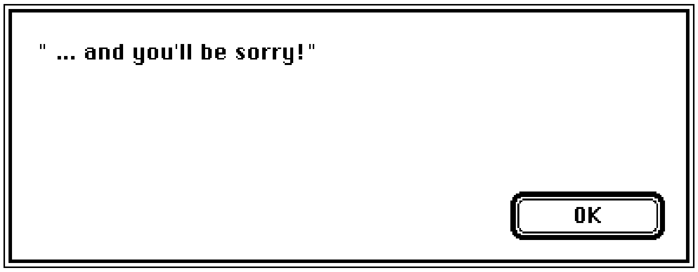
- `'" ... and you'`
- `∂'`
- `'ll be sorry!"'`

The first substring uses matching single quotation marks to quote everything from the beginning of the message up to the apostrophe.

The second substring uses the ∂ character (Option-D) to mark the apostrophe as a literal character. The third substring uses matching single quotation marks to quote the rest of the message.

The final command displays the alert box shown in Figure 3-2.

Figure 3-2 Entering the Alert command with a message



" ... and you'll be sorry!"

5. **Use the `Replace` command to replace all occurrences of one string with another string in a certain file. In your command, use the variables `{Old}` to represent the old string, `{New}` to represent the new string, and `{fName}` to represent the filename.**

   The command is

   ```
   Replace -c ∞ /{Old}/ "{New}" "{fName}"
   ```

   In the `Replace` command, the option `-c ∞` specifies that the command should be repeated until all occurrences of the old string are replaced with the new string.

After the `-c ∞` option, you type the item to be replaced within forward slashes. Then, you type the replacement string, followed by the filename. In this case, the item to be replaced, the replacement string, and the filename are all variables. You do not need to place the variable `{Old}` inside quotation marks, because the forward slashes have the effect of quoting. You do, however, need to use double quotation marks with the variables `{New}` and `{fName}`.

Using Finder Aliases in Commands

A Finder alias is an object that represents another file, directory, or volume. For example, the icon shown in Figure 3-3 is a Finder alias for the file shown in Figure 3-4.

Figure 3-3 A Finder alias

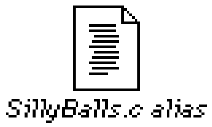
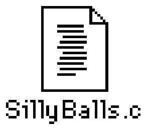


Figure 3-4 The original document the Finder alias represents



The original file, directory, or volume that a Finder alias represents is called the **target** of the Finder alias. Finder aliases are not the same as command aliases, which are alternate names you define for MPW Shell commands.

The Types of Finder Aliases

You can use Finder aliases, either by themselves or as parts of pathnames, with many MPW Shell commands. You can place a Finder alias in the middle, or at the end, of a pathname. You cannot use a Finder alias to represent a volume name. Therefore, you cannot use a Finder alias at the beginning of a full pathname, because a full pathname begins with a volume name.

When a Finder alias occurs in the middle of a pathname, it is called an **embedded alias.** When a Finder alias occurs at the end of a pathname, or as the only item in a pathname, it is called a **leaf alias.** Embedded aliases and leaf aliases behave differently in commands.

An embedded alias is always resolved and replaced by the name of its target. For example, the command

```
Catenate 'Graphics:MPW:Sample alias'
```

resolves `Sample alias` and displays the contents of Sample in the active window.

As another example, the command

```
Open 'Graphics:MPW:Examples alias:CExamples alias:SillyBalls.c'
```

resolves both `Examples alias` and `CExamples alias`, and then opens the file SillyBalls.c. An embedded alias can also have a volume such as a file server as its target. For example, the command

```
Open ':MyServer alias:TESample.c'
```

resolves `MyServer Alias`, mounts the server named MyServer, and then opens the file TESample.c.

A leaf alias is different from an embedded alias. Some commands resolve leaf aliases and some do not. For example, the command

```
Open 'Graphics:Sample alias'
```

resolves `Sample alias` and opens the file named Sample. However, the command

```
Delete 'Graphics:Sample alias'
```

does not resolve `Sample alias`. Instead, the command deletes Sample alias, leaving Sample intact.

Using the ResolveAlias Command

If you want to resolve leaf aliases in a command that does not ordinarily do so, you can resolve the alias first with the `ResolveAlias` command. For example, if you enter

```
ResolveAlias 'Sample alias'
```

the MPW Shell displays the pathname that locates the target of `Sample alias`, like this:

```
Graphics:MPW:My Work:Sample
```

To resolve an alias so that a command works on the target of the alias and not on the alias itself, use an embedded `ResolveAlias` command, as in

```
Delete `ResolveAlias Sample alias`
```

which resolves `Sample alias`, locates its target, and then deletes the target.

Commands With Special Options

Some commands have special options for using Finder aliases. For example, you can use the `-n` option with the `Exists` command to specify that leaf aliases should not be resolved. If you enter

```
Exists Sample
```

the `Exists` command checks for a file named Sample. The command

```
Exists 'Sample alias'
```

resolves `Sample alias` and also checks for a file named Sample. But if you enter

```
Exists -n 'Sample alias'
```

`Exists` checks for the file named Sample alias. To determine whether a file is a Finder alias, you can use `Exists` with the `-a` option. For example, if you enter

```
Exists -a 'WhatAmI'
```

`Exists` displays the filename WhatAmI in the active window only if the file is a Finder alias.

Another command that has special options for handling Finder aliases is `Duplicate`. By default, `Duplicate` does not resolve Finder aliases. For example, the command

```
Duplicate 'Sample alias' Graphics:MPW:
```

copies Sample alias—not Sample—to the folder named MPW on the volume named Graphics.

If, however, you use `Duplicate` with the `-rs` option, `Duplicate` resolves a Finder alias, if it appears in the command's first parameter, which is the file or directory to be copied. Thus, the command

```
Duplicate -rs 'Sample alias' Graphics:MPW:
```

resolves `Sample alias` and places a copy of the file named Sample in the MPW folder. You can also use `Duplicate` with the option `-rt` to resolve a Finder alias that appears in the command's second parameter, which is the target location.

To determine how a specific command handles Finder aliases, you can refer to its command page in *MPW Command Reference*.

Redirecting Input and Output

The data stream from which a command takes its input is known as **standard input.** By default, most MPW Shell commands use text you type in the active window as their input. The data stream a command uses for its output is known as **standard output.** Also by default, most MPW Shell commands send their output to the active window. By using the redirection characters that are described in this section, you can have a command take input from or send output to a source other than standard input and standard output.

A command also sends its error messages (sometimes called **diagnostic output**) to the active window, unless you specify otherwise. You can also use redirection characters to send error messages to a separate file or to send both the command's standard output and its error messages to one file. The data stream a command uses for its error messages is known as **standard error.**

In general, when you type a redirection character once, as with the > character, the output of the command *replaces* the contents of the target file. However, when you type a redirection character twice, as with the >> character, the output of the command *is appended to* the target file. The redirection characters are summarized in Table 3-9.

Table 3-9 The redirection characters

Character	Usage	Meaning
|	*cmd1* | *cmd2*	The output of *cmd1* becomes the input of *cmd2*.
<	< *file*	Input is taken from *file*.
>	> *file*	Output is sent to *file* and replaces its contents.
>>	>> *file*	Output is sent to *file* and is appended to its contents.
≥	≥ *file*	Error messages are sent to *file* and replace its contents.
≥≥	≥≥ *file*	Error messages are sent to *file* and are appended to its contents.
Σ	Σ *file*	Both output and error messages are sent to *file* and replace its contents.
ΣΣ	ΣΣ *file*	Both output and error messages are sent to *file* and are appended to its contents.

Piping Output to Another Command

When you use the | character between commands, the output of the first command becomes the input of the second. When you enter the line

```
Files | Count -l
```

the `Files` command generates a list of files and directories, then passes the list to the `Count` command, which counts the number of lines in the list (in other words, the number of files and directories).

Some Examples of Redirecting Input and Output

To specify that a command should read input from a source other than the active window, use the < character. For example, suppose you have a file named Messages that contains a one-line message. In the command

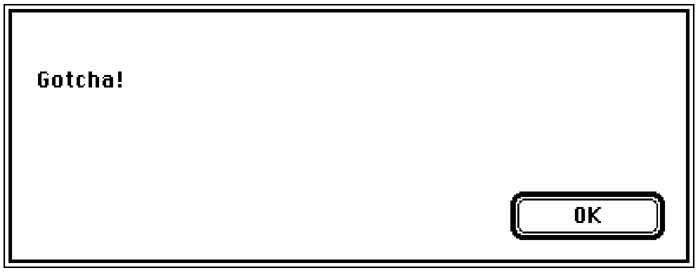
```
Alert < Messages
```

`Alert` reads the file Messages as its input, rather than taking its input from the active window, and displays an alert box containing the text in Messages. If you want to test this example, you can open a new file named Messages and then type this line in it:

```
Gotcha!
```

After you type the line, save and close the file, storing it in the MPW directory. Now enter the `Alert` command shown above. Your Macintosh beeps, and an alert box containing your message appears (Figure 3-5).

Figure 3-5 Using a file as input to the Alert command



Gotcha!

To specify that a command should send its output to a file other than the active window, use the > character or the >> character. For example, if your MPW directory contains the files File1 and File2, the command

```
Catenate File1 File2 > File3
```

concatenates File1 and File2 and stores them in a file named File3. If File3 does not exist, the command creates it. If File3 does exist, its contents are replaced with the new output. The original copies of File1 and File2 are not affected.

If you enter the same command using the >> character, as in

```
Catenate File1 File2 >> File 3
```

File1 and File2 are concatenated and stored in File3. If File3 does not exist, the command creates it. If File3 does exist, the command adds its output to the end of File3. In this case also, the original copies of File1 and File2 are not affected.

Some Examples of Redirecting Error Messages

By default, a command sends its error messages to the active window. However, sometimes you want the command to send its error messages to a separate file. To redirect error messages, use the ≥ character (Option-period) or the ≥≥ character (Option-period twice). For example, the command

```
Catenate File1 File2 > File3 ≥ Errors
```

concatenates File1 and File2, sends the concatenated version to File3, and sends any error messages to a file named Errors. If Errors does not exist, the command creates it. If Errors does exist, its contents are replaced with the new error messages.

To append error messages to a file, instead of replacing a file's contents, use the ≥≥ character, as in

```
Catenate File1 File2 > File3 ≥≥ Errors
```

which concatenates File1 and File2, sends the concatenated version to File3, and sends any error messages to the file named Errors. If Errors does not yet exist, the command creates it. If Errors does exist, the command adds its error messages to the end of the file.

Often it is useful to store all of a command's output, including error messages, in the same file. To do so, you would use the ∑ character (Option-W). For example, if you enter the command

```
Asm -a Sample.a ∑ Testfile
```

the MPW Shell sends its output and error messages to Testfile, creating Testfile if it does not exist, and replacing its contents with the new output if it does exist.

You would use the ∑ character when you want to redirect all of a command’s output, including both standard output and error messages, to one place. Note that a command like

```
Asm -a Sample.a > Testfile ≥ Testfile
```

which attempts to send its standard output and then its error messages to one file, does not work because a command cannot open the same file twice.

When you use the ∑∑ character (Option-W twice), both the output and error messages are appended to the file you name. Therefore, a command like

```
Asm -a Sample.a ∑∑ Testfile
```

appends both its output and its error messages to Testfile. If Testfile does not exist, the command creates it. If Testfile does exist, the command adds its output to the end of the file.

Some Examples of Redirection With Finder Aliases

When you use a Finder alias in a command that redirects input, output, or error messages, the Finder alias is *always* resolved. For example, if you create a file named Messages with a one-line message, you can create a Finder alias of Messages named Messages alias. Then, if you enter

```
Alert < 'Messages alias'
```

the `Alert` command resolves `Messages alias`, opens the file Messages, and displays an alert box containing the message. (The alert box would look like the one in Figure 3-5 on page 3-29.)

Likewise, if your MPW directory contains the files File1 alias and File2 alias, you could enter the command

```
Catenate 'File1 alias' 'File2 alias' > File3
```

to store the contents of the files File1 and File2 in the file named File3.

Redirecting Input and Output to Virtual Devices

Sometimes you want to enter a command that takes input from or sends output to a virtual device, rather than a physical device. A **virtual device** is a source of input or a destination for output that does not have a corresponding physical device. For example, video monitors, disk drives, and printers are all physical devices. However, standard input and standard output are virtual devices because they do not have a corresponding physical location.

MPW Shell commands can take input from or send output to the virtual devices listed in Table 3-10.

Table 3-10 The virtual devices you can use in MPW Shell commands

Virtual devices	Description
`Dev:Console`	The window from which the command was executed
`Dev:StdIn`	The current standard input
`Dev:StdOut`	The current standard output
`Dev:StdErr`	The current diagnostic output
`Dev:Null`	A null device, used to discard unwanted output

You can use the names of virtual devices in a command just as you would use a filename. For example, a command might produce error messages or other unwanted output that you want to discard. You can enter

```
Open "Sample 1" ≥≥ Dev:Null
```

to discard any error messages the `Open` command might produce.

CHAPTER 4

Working With Files and Directories

Contents

Working with source files is basic to working with the MPW Shell because source files are the building blocks of your development projects. You will find powerful editing features in both the MPW Shell menus and the MPW Shell editing commands.

This chapter describes how to use the MPW Shell menus to create and edit source files and how to specify those files in MPW Shell commands. Specifically, this chapter explains how to

- name source files and use stationery pads
- use each of the formatting features the MPW Shell provides
- set default format values for your MPW environment
- write pathnames that locate files

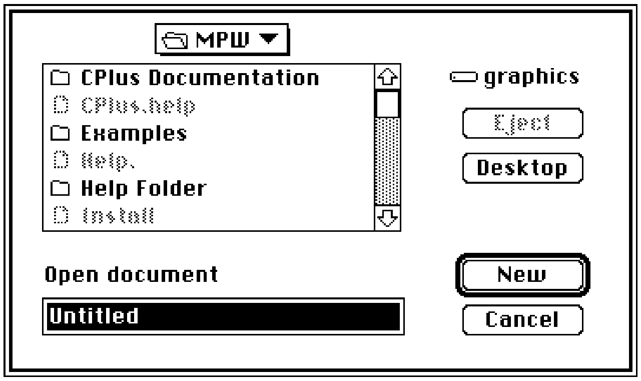
Before you start this chapter, you should understand all of the topics presented in the first three chapters of this guide. Once you complete this chapter, you can proceed to "Editing With Commands."

Creating Source Files

An MPW Shell source file is simply a text file that you create and format with the text editor. You can do much of the work of creating and editing source files with the items in the File, Edit, Find, and Mark menus.

To create a new MPW Shell file, you choose New from the File menu. A dialog box appears (Figure 4-1) asking you to name the document.

Figure 4-1 Creating an MPW Shell document



Naming an MPW Shell Document

When you name an MPW Shell file, you can

- include up to 31 characters
- use any character except a colon
- use any combination of uppercase and lowercase letters

If you use a space or one of the MPW Shell special characters in a filename, you must quote the name properly whenever you use it in a command. To check whether a character has a special meaning to the MPW Shell, you can refer to the appendix “Table of Special Characters and Operators.”

Whenever possible, try not to use spaces or special characters in filenames, unless you use a period, an underscore, or a • character (Option-8). For example, instead of naming a file Chapter 1, you could use any of these names:

```
Chapter1
Chapter.1
Chapter_1
Chapter•1
```

None of these names require quotation marks in commands.

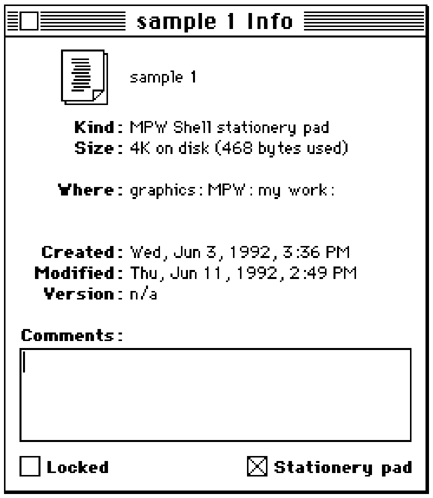
However, if you must use a space or a special character in a filename, you should first understand the rules of using quotation marks in MPW Shell commands, which are explained in the section “When to Use Quotation Marks in Commands” on page 3-15.

Creating and Using Stationery Pads

You can save any MPW Shell file as a **stationery pad,** that is, as a document that serves as a template from which you create other files. Stationery pads are useful for information that is repeated in many documents, such as header files, main programs, or assembly-language procedures.

To make an MPW Shell document a stationery pad, you click the document’s icon, then choose Get Info from the Finder’s File menu. When the Get Info dialog box appears (Figure 4-2), click the “Stationery pad” checkbox in the lower-right corner.

Figure 4-2 Making a document a stationery pad



sample 1 Info

sample 1

Kind: MPW Shell stationery pad
Size: 4K on disk (468 bytes used)

Where: graphics:MPW:my work:

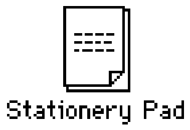
Created: Wed, Jun 3, 1992, 3:36 PM
Modified: Thu, Jun 11, 1992, 2:49 PM
Version: n/a

Comments:

☐ Locked ☒ Stationery pad

Once a document is a stationery pad, its icon changes to look like the one in Figure 4-3.

Figure 4-3 A stationery pad icon



When you open a stationery pad, a dialog box appears in which you can name the new file you are creating from it (see Figure 4-1). The new file contains the contents of the stationery pad, but leaves the stationery pad unchanged.

Navigating Through Files

The MPW Shell provides special keystrokes you can use for navigating through long files, selecting text, and deleting text quickly.

The keystrokes you use to navigate through text are listed in Table 4-1. Two of the most useful keystrokes are Command–Option–Up Arrow, which moves to the beginning of a file, and Command–Option–Down Arrow, which moves to the end.

Table 4-1 Shortcuts for moving through text

Keystroke	How the insertion point moves
Up Arrow	Moves up one line
Down Arrow	Moves down one line
Left Arrow	Moves left one character
Right Arrow	Moves right one character
Option–Left Arrow	Moves left one word
Option–Right Arrow	Moves right one word
Command–Down Arrow	Moves down one screen
Command–Up Arrow	Moves up one screen

Table 4-1 Shortcuts for moving through text

Keystroke	How the insertion point moves
Command–Left Arrow	Moves to the beginning of the line
Command–Right Arrow	Moves to the end of the line
Command–Option–Up Arrow	Moves to the beginning of the file
Command–Option–Down Arrow	Moves to the end of the file
Command–Shift–Up Arrow	Selects the text between the insertion point and the beginning of the file
Command–Shift–Down Arrow	Selects the text between the insertion point and the end of the file

As you work with MPW Shell text files, there are a number of shortcuts you can use for selecting text; these are listed in Table 4-2.

Table 4-2 Shortcuts for selecting text

Action	What it selects
Double-click	A word
Triple-click	A line
Double-click on `( ) [ ] { } ' " / \ `` `	Everything between the character and its mate
Click, then move the insertion point and Shift-click	Everything between the two clicks

Deleting text in an MPW Shell file is simple and intuitive. To delete a character, you usually press the Delete key; to delete a block of text, you select it and press Delete. The MPW Shell provides some special keystrokes for deleting text quickly; these are listed in Table 4-3.

Table 4-3 Shortcuts for deleting text

Keystroke	What it deletes
Delete	Deletes the character to the left of the insertion point
Del*	Deletes the character to the right of the insertion point
Option-Delete	Deletes the word to the left of the insertion point
Option-Del*	Deletes the word to the right of the insertion point
Command-Delete	Deletes from the insertion point to the end of the file

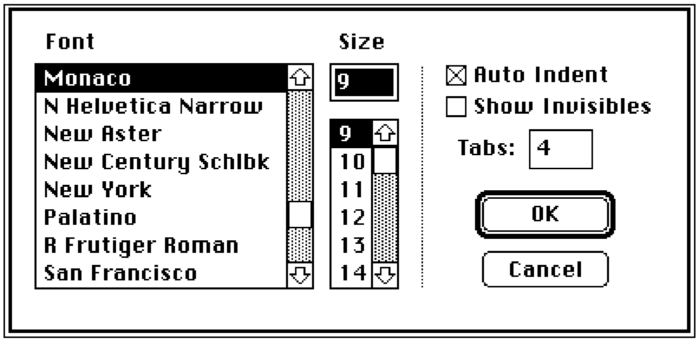
* This key is on extended keyboards only, on the keypad between the alphabet keys and the number keys.

Changing the Format of Files

When you open a new MPW Shell document and begin typing, your text appears in 9-point Monaco font. Because Monaco is a monospaced font, it is useful for entering source code.

To change to a new font or size, you can choose Format from the Edit menu to open the Format dialog box (Figure 4-4).

Figure 4-4 The Format dialog box



From the Format dialog box, you can select a new font or font size, change the spacing of tabs, indent text automatically, and display the invisible nonprinting characters in a file.

Note
For source files, it is best to use a monospaced font such as Monaco or Courier because source code typed in a proportionally spaced font can be hard to read. ◆

Indenting Text Automatically

Automatic indentation is useful for entering blocks of source code because it allows you to type a group of lines that are indented by the same amount of space. For example, suppose you have turned on the Auto Indent option and you want to type these lines:

```
void Initialize()
{
    WindowPtr     mainPtr; /* Press Tab to begin this line */
    OSErr         error;   /* No need to press Tab here */
    SysEnvRec     theWorld;         /* But press Option-Return here */
```

Automatic indentation is turned on by default. If you want to type the lines shown above, you would press the Tab key once, at the beginning of the `WindowPtr` line. All of the following lines align with the `WindowPtr` line, until you press Option-Return to return to the left margin. To turn off automatic indentation, you open the Format dialog box and click the Auto Indent checkbox to deselect it.

Displaying Nonprinting Characters

At times it is useful to display the nonprinting characters in a text file, for example, to check the number of return, tab, or space characters in a line, or to track down stray control characters that might have entered a file.

To display **nonprinting characters,** also known as invisible characters, you would go to the Format dialog box, then click the Show Invisibles checkbox. The nonprinting characters that are displayed have the following meanings:

∆	Tab
◊	Space
¬	Return
¿	All other control characters

Shifting Text to the Left or Right

The Shift Left and Shift Right menu items can help you enter source code easily. For example, you might enter some lines of source code quickly, as in the following lines:

```
{
NumToString (num, outStr);
TextFace ();
DrawString (outStr);
TextFace (Bold);
```

If you select the last four lines and choose Shift Right from the Edit menu, those lines would shift to the right by one tab stop, like this:

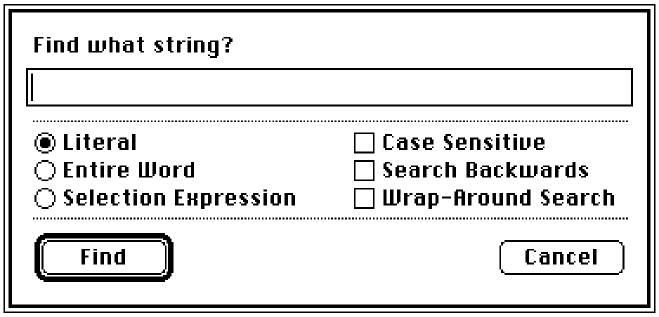
```
{
    NumToString (num, outStr);
    TextFace ();
    DrawString (outStr);
    TextFace (Bold);
```

To move a block of text to the left or right, select the text, then choose Shift Left or Shift Right from the Edit menu. The text moves by one tab setting, which has the number of spaces indicated in the Tabs box of the Format dialog box. To move the text another tab stop to the left or right, choose Shift Left or Shift Right again.

Finding and Replacing Text

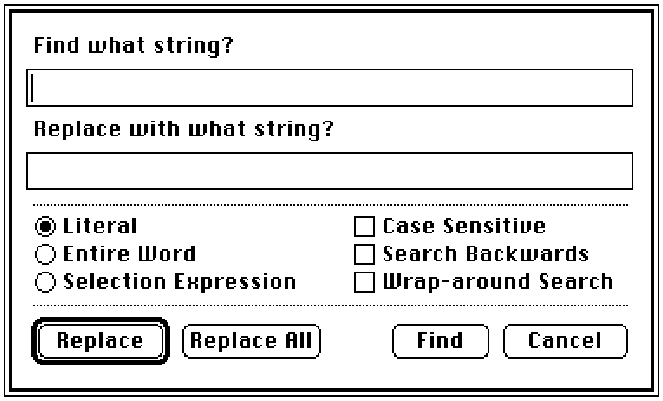
You can use the Find menu to find, or to find and replace, a text string or a pattern. Choosing Find from the Find menu opens the Find dialog box (Figure 4-5), while Choosing Replace from the Find menu opens the Replace dialog box (Figure 4-6).

Figure 4-5 The Find dialog box



● Literal
○ Entire Word
○ Selection Expression
☐ Case Sensitive
☐ Search Backwards
☐ Wrap-Around Search

Figure 4-6 The Replace dialog box



Using the Find and Replace dialog boxes is easier than entering a complex command to find or replace a text string or expression. You can enter a string or a pattern in the "Find what string?" or "Replace with what string?" text field.

When you enter a pattern, you should place it within forward or backward slashes, depending on whether you want to search forward or backward from the insertion point. Once you enter the pattern, you click Selection Expression, then Find or Replace.

For example, if you enter

```
/bo?/
```

in the "Find what string?" text field, and then click the Selection Expression button and the Find button, the first string that matches the expression (for example, `Bob`) is selected. Likewise, if you enter

```
\bo?\
```

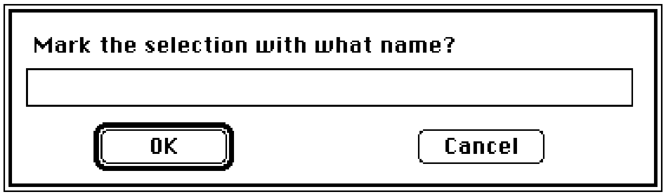
in the "Find what string?" text field, the search proceeds backward from the insertion point to locate any strings that match the expression.

You can enter many types of selection expressions in the Find and Replace dialog boxes. Selection expressions are described in more detail in the chapter "Editing With Commands."

Adding and Removing Markers

You can add an invisible marker at any place in a text file so that later you can easily find that spot. Markers provide a convenient way to identify and locate parts of your source code files. To add a marker to a file, you place the insertion point at the location you want to mark, and then choose Mark from the Mark menu. When the dialog box appears (Figure 4-7), you can enter a name for your marker.

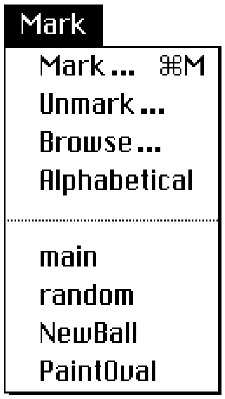
Figure 4-7 Adding markers to a file



Markers are case sensitive. For example, a marker named PaintOval is different from a marker named paintOval. Markers are an exception to the general rule that most MPW Shell commands, strings, and names are not case sensitive.

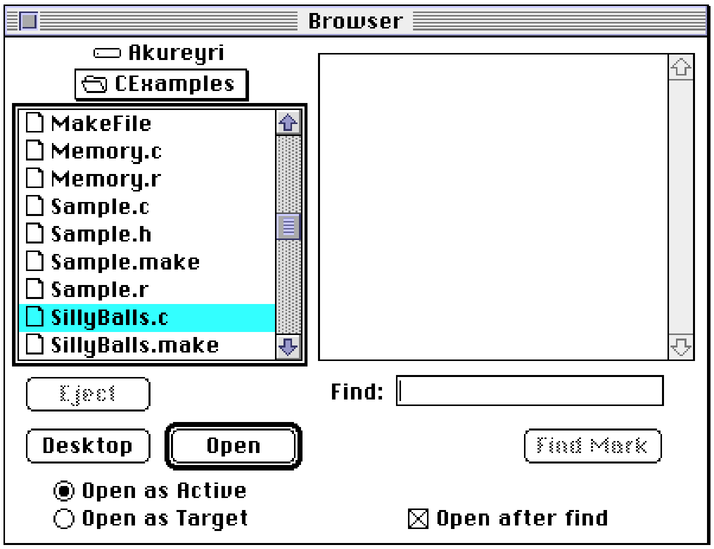
Once you add a marker to a file, it appears in the bottom half of the Mark menu when- ever the file is open, as shown in Figure 4-8 on page 4-14. In the Mark menu, the markers appear in the order in which they exist in the file, unless you choose Alphabetical to arrange them alphabetically. Once you set a marker, it remains with its text unless you delete the text or cut it and paste it elsewhere.

Figure 4-8 The Mark menu, after markers are added



You can search for markers without opening a file by choosing Browse from the Mark menu. When you do, the Browser dialog box opens (Figure 4-9). To display the markers in a file, select the file from the list on the left. When you do, the markers in the file are displayed in the list at the right. When you choose a marker from the list, then click Open, the file opens and scrolls to the marker you selected.

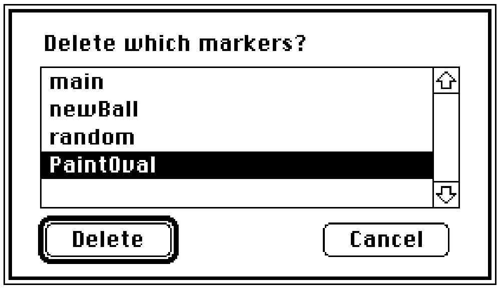
Figure 4-9 The Browser dialog box



If you know a marker name but don't know where it is located, you can enter the marker name in the Find text field, click the "Open after find" checkbox, and then click the Find Mark button to open the file and scroll to the marker.

To remove a marker from a file, open the file, then choose Unmark from the Mark menu. When the dialog box appears (Figure 4-10), select the marker you want to remove, then click Delete. When you delete a marker, its name is removed from the Mark menu.

Figure 4-10 Removing markers


Delete which markers?

main
newBall
random
PaintOval

Tutorial—Specifying Default Formats

Rather than setting text formats each time you use the MPW Shell, you can set them as defaults so that they are initialized each time you start MPW. This tutorial describes how to set formats with the `UserVariables` Commando dialog box.

The commands you build that set formats are displayed in the Worksheet window. The new formats take effect until you exit from the MPW Shell. If you want to make them more permanent, you can select the commands that appear in the Worksheet window and add them to your UserStartup•*name* file.

In this tutorial, you will

- open the `UserVariables` Commando dialog box
- build a command that sets a default font size
- build a command that causes text searches to wrap
- build a command that sets default values for printing files
- execute the resulting commands so that they take effect for this MPW session

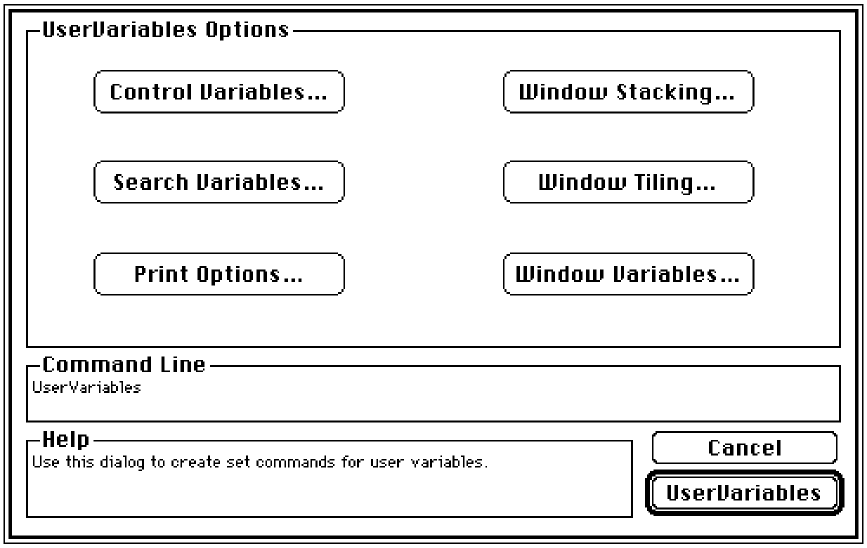
To open the `UserVariables` Commando dialog box and specify default formats, follow these steps:

1. **Type this command in the Worksheet window, then press Enter:**

   ```
   UserVariables
   ```

   The `UserVariables` Commando dialog box opens (Figure 4-11).

Figure 4-11 The UserVariables Commando dialog box



UserVariables

Use this dialog to create set commands for user variables.

Remember that `UserVariables` is an exception; to open its Commando dialog box so that it displays a command in the active window, you type `UserVariables`, then press Enter.

Notice that the Command Line box reads

```
UserVariables
```

2. **Click Window Variables.**

   The Window Variables dialog box appears (Figure 4-12).

   In the Window Variables dialog box, you can specify a default font, font size, or tab size and you can turn automatic indentation on or off.

3. **Type `10` in the FontSize box.**

   The command

   ```
   Set FontSize 10
   ```

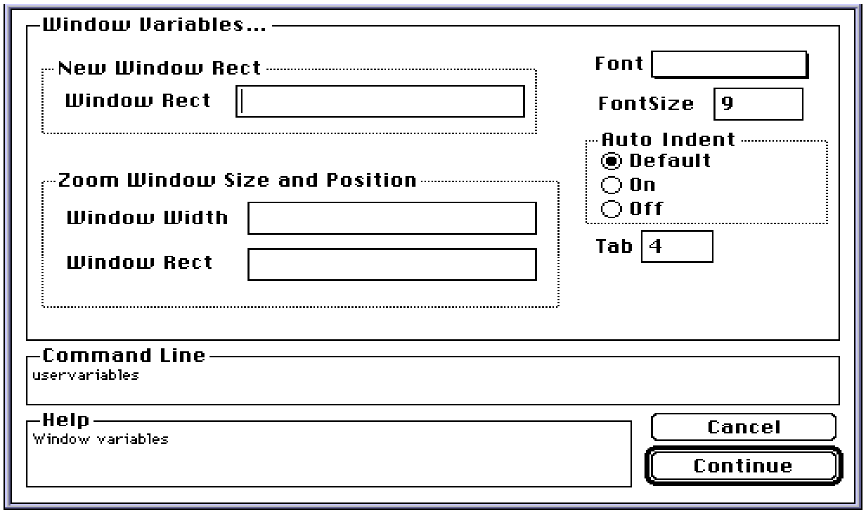
   is added to the Command Line box. Later, when you add this command to the Worksheet window and execute it, the text you enter in new windows will appear in 10-point Monaco, which is a bit easier to read than the default 9-point Monaco.

This command does not affect the text that is already in the Worksheet window.

4. **Click Continue.**

   You return to the `UserVariables` Commando dialog box.

Figure 4-12 The Window Variables dialog box



FontSize 9

Auto Indent: [✓ Default] On Off

Tab 4

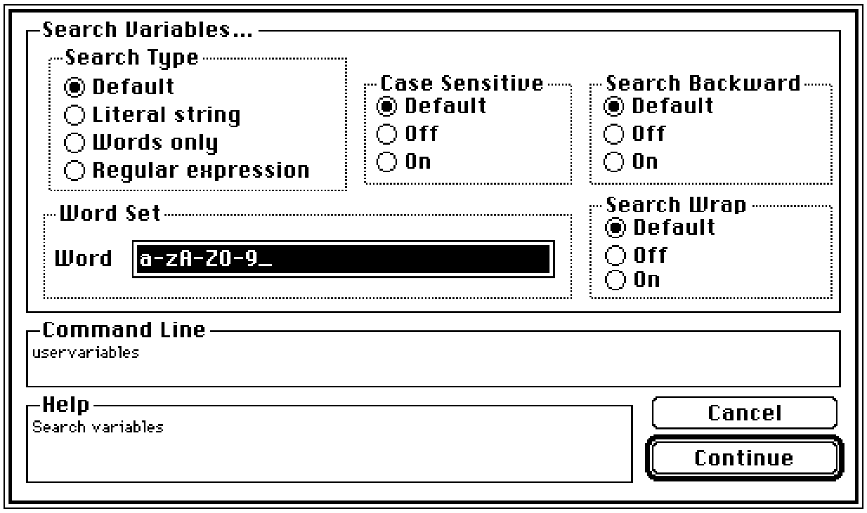
Command Line: uservariables

Help: Window variables

5. **Click Search Variables.**

   The Search Variables dialog box appears (Figure 4-13).

Figure 4-13 The Search Variables dialog box


Search Variables...

Search Type: (● Default) ○ Literal string ○ Words only ○ Regular expression

Case Sensitive: (● Default) ○ Off ○ On

Search Backward: (● Default) ○ Off ○ On

Word Set — Word: a-zA-Z0-9_

Search Wrap: (● Default) ○ Off ○ On

Command Line
```
uservariables
```

Help: Search variables

6. **Click the On button in the Search Wrap box.**

   You have just turned on Search Wrap, so that you can begin searching for a text string from the middle of a file. When the search reaches the end of the file, it begins searching again from the beginning of the file.

   The command in the Command Line box changes to

```
UserVariables ; Set SearchWrap 1 ; Set FontSize 10
```

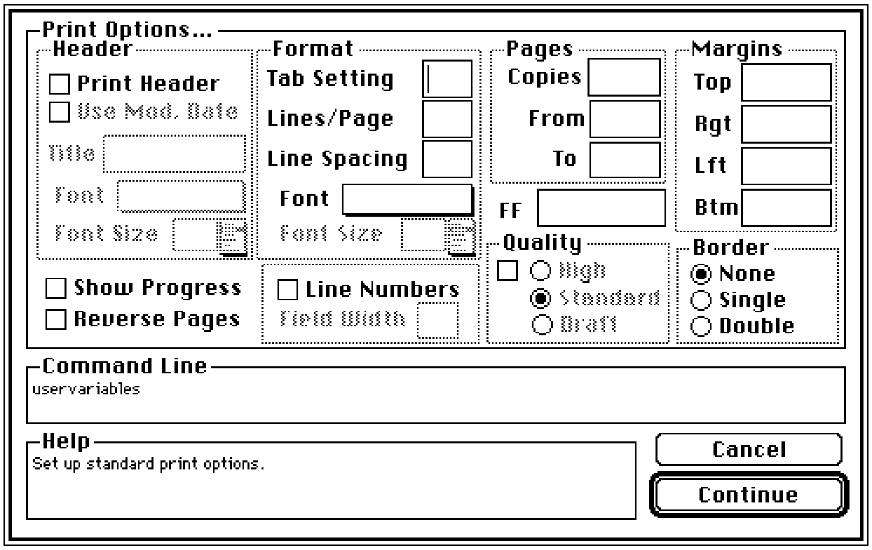
7. **Click Continue.**

   You return to the `UserVariables` Commando dialog box again.

8. **Click Print Options.**

   The Print Options dialog box appears (Figure 4-14).

Figure 4-14 The Print Options dialog box



Quality: High [✓ Standard] Draft

Border: [✓ None] Single Double

Command Line: userverariables

Help: Set up standard print options.

At this point, you can choose or enter options that take effect each time you print. The options outlined in steps 9 through 11 of this tutorial are suggestions; you can choose whether or not to enter them.

9. **Click the Show Progress checkbox.**

   When Show Progress is turned on, the MPW Shell displays messages in the active window that show which file is currently printing and the number of lines and pages printed.

10. **Click the Line Numbers checkbox.**

    When Line Numbers is turned on, each line of the printed text is numbered at the left margin. Line numbers are useful for debugging source files.

11. **For the Margins options, type**

    - `1` for Top
    - `.5` for Rgt
    - `.5` for Lft
    - `1` for Btm

The text fields of the Margins options specify the margins on the printed page, in inches. The Command Line box now contains the commands

```
UserVariables ; Set SearchWrap 1 ; Set FontSize 10 ; ∂
Set PrintOptions '-p -tm 1 -lm .5 -rm .5 -bm 1 -n '
```

Here, the command is written on two lines, with the ∂ character at the end of the first line. In the Command Line box, the command takes just one line, so the ∂ character is not necessary.

12. **Click Continue.**

    You return to the `UserVariables` Commando dialog box.

13. **Click UserVariables.**

    The commands you have built are displayed in the Worksheet window.

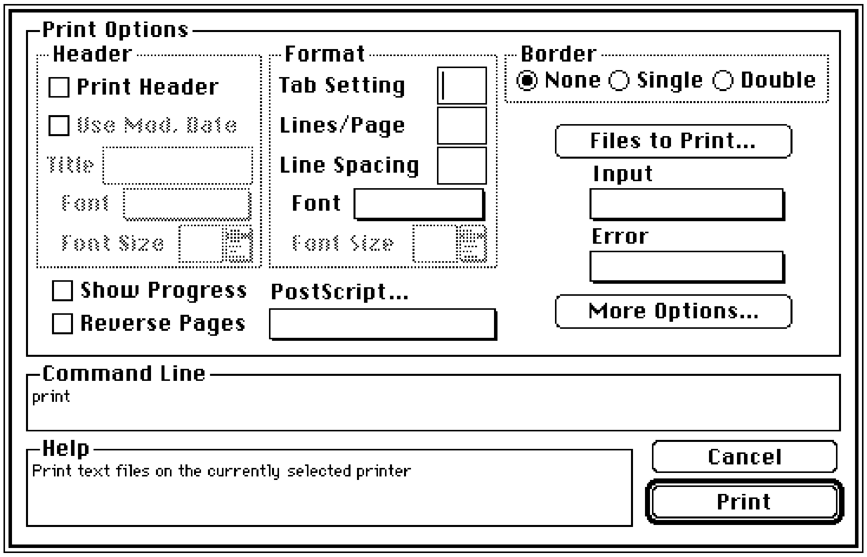
14. **To execute the commands, select everything from the first `Set` command to the end of the line, then press Enter.**

Once you execute the commands, they take effect for your current MPW session so that you can experiment with the values you have set. The commands remain in the Worksheet window. If you like, you can add them to your UserStartup•*name* file to make them permanent and have them initialized each time you start MPW. You can find a sample UserStartup•*name* file in the appendixes.

Printing Files

You can use menu items and Commando dialog boxes to print entire files, sections of files, or the portion of a file that appears in a window. The easiest way to print an entire file is with the `Print` Commando dialog box (Figure 4-15).

Figure 4-15 The Print Commando dialog box



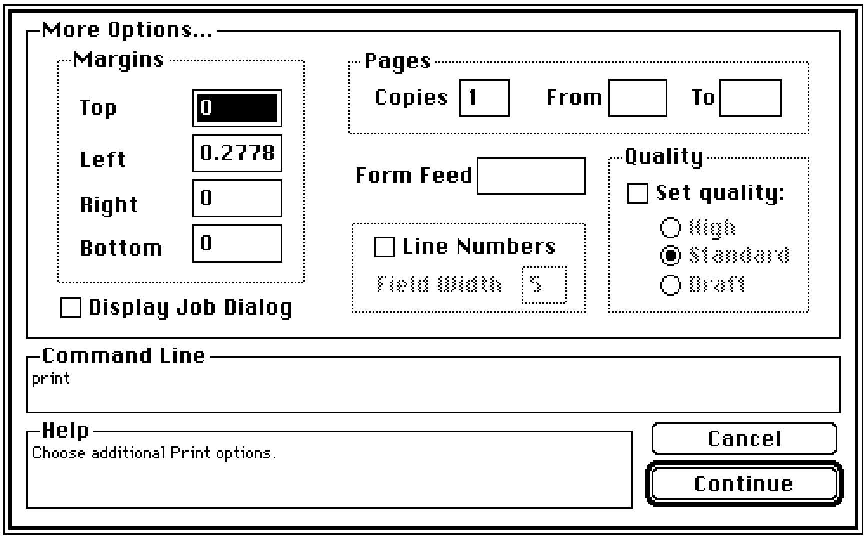
```
print
```

Print text files on the currently selected printer

Note
You can also use the `Print` command to print files. The `Print` command is not described in detail in this guide, but is described in *MPW Command Reference*. ◆

The `Print` Commando dialog box offers many useful printing options. When you click the More Options button, a second dialog box appears (Figure 4-16) that gives you more options.

Figure 4-16 More Options from the Print Commando dialog box



```
print
```

Choose additional Print options.

From these two dialog boxes, you can specify

- a list of files to print
- a page header for the printed copy
- formatting options, such as tabs, line spacing, font, and margins
- the number of copies to print
- a range of pages to print
- line numbers on the printed copy
- where page breaks should be inserted
- the name of a file that contains PostScript™ commands that affect the printed copy

Inserting Page Breaks in a File

One important feature of the More Options dialog box (Figure 4-16) is the Form Feed box. You can use the Form Feed box to specify a text string in a file that causes a page break. You add the string to the file to mark the page break; and when you print the file, you enter the string in the Form Feed box.

If your file is a source file, the string you enter should be a comment so that it is not compiled or assembled as part of the file. To write the comment, use the syntax that is appropriate to the programming language you use. For example, any of these strings might be valid comments, depending on the programming language you use:

```
/* ff */
// ff
{ff}
(* ff *)
```

For more information on writing comments, refer to your language reference manual.

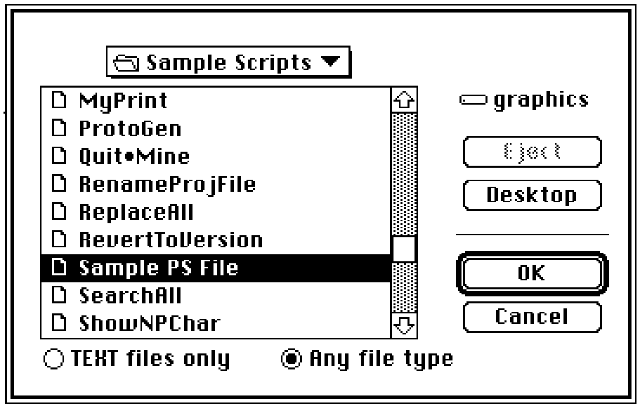
Specifying a PostScript File

In the `Print` Commando dialog box (Figure 4-15), the pop-up menu labeled PostScript allows you to specify the name of a file written in the PostScript language that affects the appearance of the printed output. The commands in the PostScript file are sent to the printer before each page of the printed output is sent.

For example, a PostScript file might specify a font, the position of the text block on the page, the depth of color of the text, or any other features that PostScript offers. You can find a sample PostScript file, named `Sample PS File`, in the MPW Script Tips folder. Using the PostScript pop-up menu is the equivalent of using the `-ps` option with the `Print` command.

To specify a PostScript file in the `Print` Commando dialog box, you choose Input File from the PostScript pop-up menu. When the standard file dialog box appears (Figure 4-17), you can locate and select the PostScript file you want to use.

Figure 4-17 Choosing a PostScript file

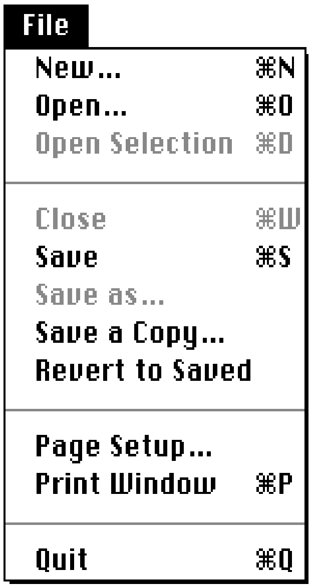


Note
If you specify a PostScript file to be used when you print a document, be sure to print the document on a printer that uses the PostScript language, such as one of the Apple LaserWriter printers that has PostScript installed. ◆

Printing Sections of Files

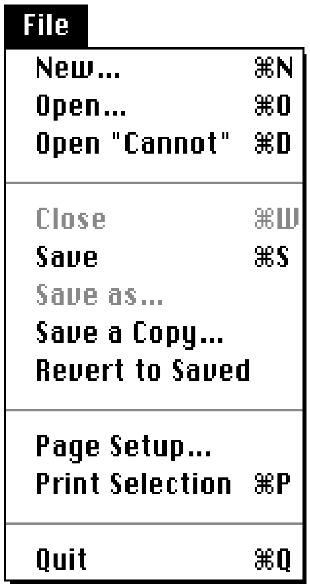
Sometimes, especially when you are debugging a program, it is useful to print only a portion of a file. The File menu (Figure 4-18) has two menu items you can use, Print Window and Print Selection.

Figure 4-18 The File menu, with the Print Window item



Ordinarily, when you have a window displayed, the File menu includes the Print Window menu item. Print Window allows you to print all of the text that is displayed in the active window. When you select some text in the window, the menu item changes to Print Selection (Figure 4-19), which allows you to print only the text that is selected.

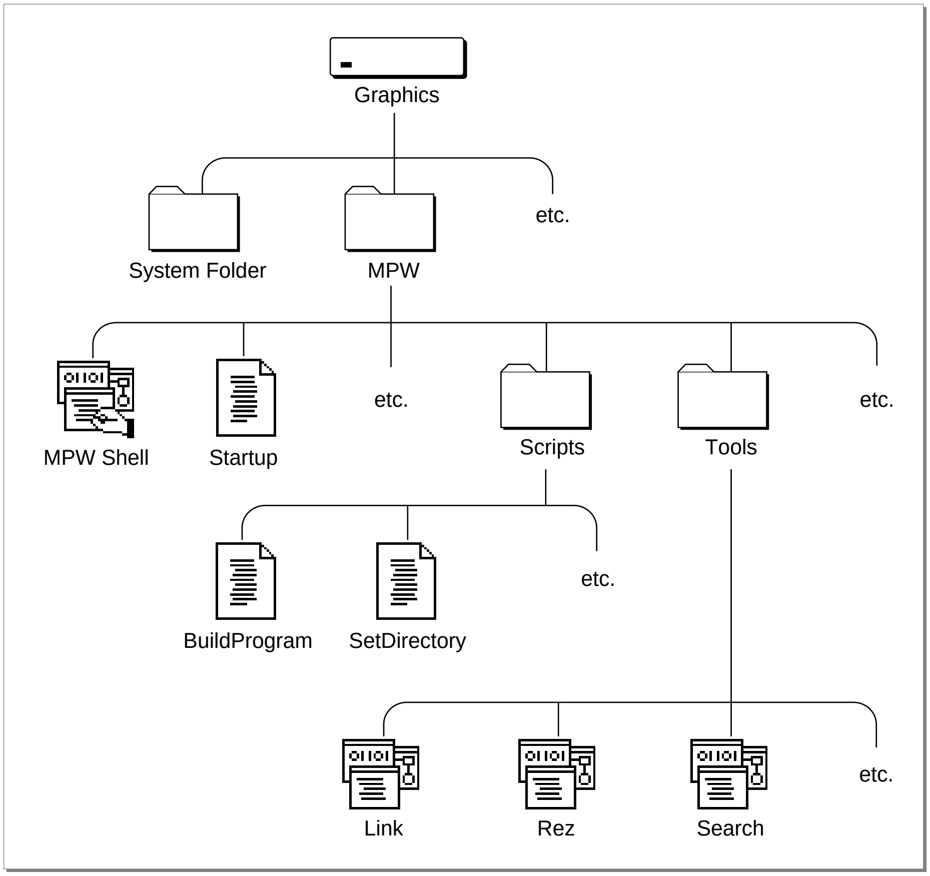
Figure 4-19 The File menu, with the Print Selection item



Organizing Files Into Directories

On the Macintosh, as on many computers, files are organized in a hierarchical structure, as illustrated in Figure 4-20 on page 4-22. The icons that depict a hard disk, folders, and documents represent a volume, directories, and files.

Figure 4-20 The hierarchical structure of the file system



You can store files directly on a volume or within a directory, and you can nest directories within other directories. Each file has a full name, called a full pathname, that shows the route to the file from the volume, through all of the necessary directories, to the filename. Pathnames are necessary to locate files because you can have more than one file of the same name stored in different directories.

Each part of the pathname is separated from the next by a colon. For example, the pathname to the Search file (which is actually the MPW Shell `Search` tool) shown in Figure 4-20 is

```
Graphics:MPW:Tools:Search
```

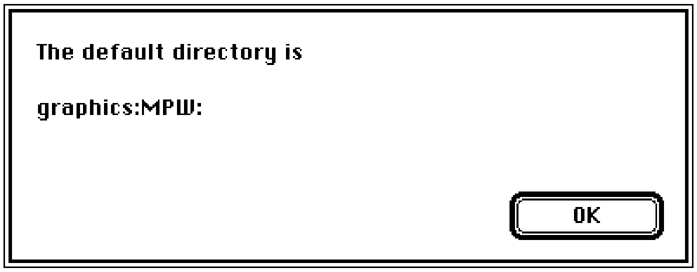
The pathname shows that the `Search` tool is stored in the Tools directory; the Tools directory is stored in the MPW directory; and all three objects are stored on the volume named Graphics.

Setting the Current Directory

The **current directory** is the directory in which you create and save files. As you work with the MPW Shell, you can have many files from different directories open at once. The current directory is not necessarily the directory that contains the file you are working on. The current directory is the directory you set with the Directory menu or with the `Directory` command. (The method described in this section is the Directory menu. You can find more information on the `Directory` command in the *MPW Command Reference*.)

When you first start MPW, the current directory is MPW. To check which directory is presently the current directory, you choose Show Directory from the Directory menu. A dialog box showing the full pathname of the current directory appears (Figure 4-21).

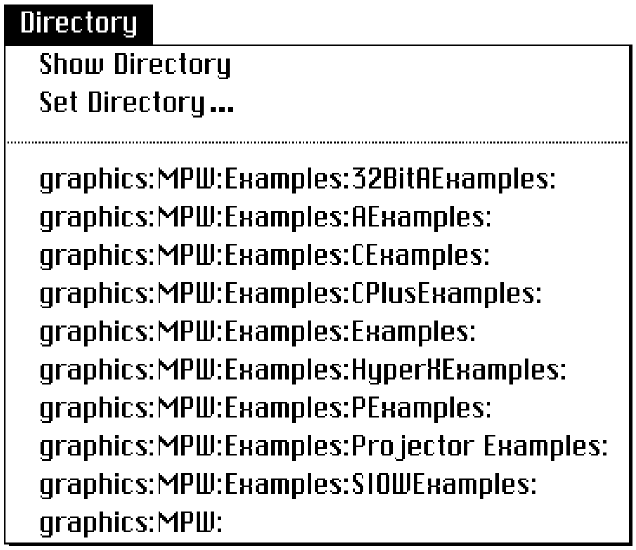
Figure 4-21 Displaying the name of the current directory


The default directory is

graphics:MPW:

To change the current directory, you use the Directory menu (Figure 4-22).

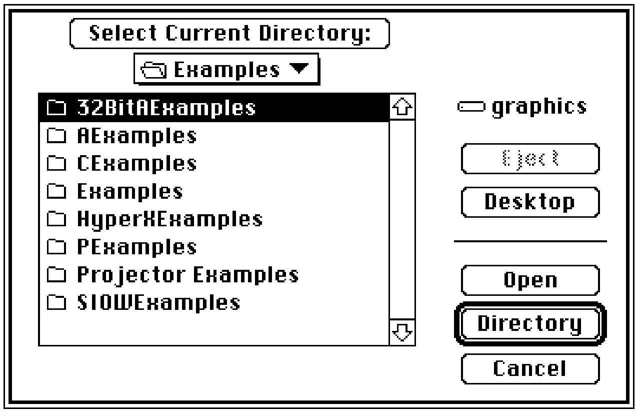
Figure 4-22 Choosing a directory from the Directory menu


Directory
Show Directory
Set Directory…
graphics:MPW:Examples:32BitAExamples:
graphics:MPW:Examples:AExamples:
graphics:MPW:Examples:CExamples:
graphics:MPW:Examples:CPlusExamples:
graphics:MPW:Examples:Examples:
graphics:MPW:Examples:HyperXExamples:
graphics:MPW:Examples:PExamples:
graphics:MPW:Examples:Projector Examples:
graphics:MPW:Examples:SIOWExamples:
graphics:MPW:

The bottom part of the Directory menu lists the full pathnames of the directories you use frequently. (You can also add a directory to the Directory menu by entering a command in your UserStartup•*name* file. For more information, refer to the sample UserStartup•*name* file in the appendixes.)

If the directory you want is listed there, you can simply choose it from the menu. If it isn't, you can choose Set Directory. When you choose Set Directory, a standard file dialog box appears, from which you can choose any directory on your system (Figure 4-23).

Figure 4-23 Choosing a directory from a standard file dialog box



To use the dialog box, you can do one of the following:

- Select a directory in the list and then click Directory.
- Select a directory from the pop-up menu above the list, then click Select Current Directory.

Once you set the current directory, files are stored there by default. You can also write shorter pathnames that begin with the current directory, rather than with the volume name.

Writing Full and Partial Pathnames

A pathname that starts partway down the path, with the current directory, is called a **partial pathname.** A partial pathname begins with a colon, which represents the current directory. For example, suppose you have a directory structure like the one shown in Figure 4-24.

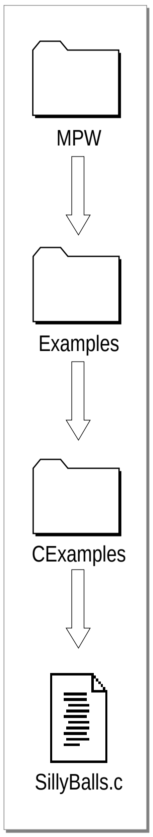
If the current directory is Examples, all you need to write to locate the file SillyBalls.c is

```
:CExamples:SillyBalls.c
```

rather than the full pathname, which might be

```
Graphics:MPW:Examples:CExamples:SillyBalls.c
```

Figure 4-24 Part of a directory structure



Double colons (`::`) in a pathname specify the directory one level above the current directory, which is known as the **parent directory.** For example, if the current directory is Examples, the pathname

```
::Examples:CExamples:SillyBalls.c
```

starts with MPW, the directory one level up from Examples. If the current directory is Examples, you can also enter

```
Files ::
```

to get a list of the files in the directory one level up from Examples, which is MPW.

Triple colons (`:::`) in a pathname specify the directory two levels above the current directory, called the **grandparent directory.** For example, if the current directory is CExamples, the pathname

```
:::CExamples:SillyBalls.c
```

starts with MPW, which is two levels up from CExamples. In this case, the three colons at the beginning of the pathname represent the MPW directory. If the current directory is CExamples, you can enter

```
Files :::
```

to get a list of files in the grandparent directory, which is MPW.

When the file you are interested in is located in the current directory, you can specify it using a **leafname,** which contains just the filename, with no colons. For example, all of the following names are leafnames:

```
SillyBalls.c
Date
Rez
```

For example, if the current directory is CExamples, and you want to open the file named SillyBalls.c, you can enter either of the commands

```
Open :SillyBalls.c
Open SillyBalls.c
```

In the first command, `:SillyBalls.c` is a partial pathname. In the second command, `SillyBalls.c` is a leafname. Either command would open the file named SillyBalls.c.

Using Variables in Pathnames

Variables in a pathname make the pathname shorter and easier to write. For example, the variable `{MPW}` contains the full pathname of the MPW directory, starting with the volume name. To specify the Tools directory that is stored within the MPW directory, you could use

```
{MPW}Tools
```

rather than entering the full pathname, which might be

```
Graphics:MPW:Tools
```

The variable `{MPW}` contains the full pathname of the MPW directory, including the colon following MPW. After `{MPW}`, you need only enter the part of the pathname from the MPW directory to the file or directory you want. The names of the variables built into the MPW Shell that you can use in commands are listed in Table 3-3 and Table 3-4 in the chapter “Entering Commands.”

You can also specify the invisible Desktop Folder directory in a pathname (the Desktop Folder is a directory that contains information about the icons stored on the desktop). To use the Desktop Folder in a pathname, you use the variable `{Boot}`, which contains the name of the volume on which your system software is stored. The Desktop Folder is a directory of `{Boot}`. To specify items stored on the desktop in an MPW Shell command, use a pathname like this one:

```
"{Boot}Desktop Folder:MPW Shell alias"
```

The last pathname locates a Finder alias of the MPW Shell that has been stored on the desktop. (Finder aliases are discussed in more detail in the section “Using Finder Aliases in Commands” on page 3-25.) Notice that the pathname is placed within double quotation marks because it contains both a variable and a filename with spaces.

Writing a Pathname for an Application

From the MPW Shell, you can execute any application on your hard disk by entering the application’s name as a command. For example, if you have an application named ResEdit stored in the MPW directory, and if the current directory is MPW, you can start ResEdit by entering

```
ResEdit
```

in the active window. If MPW is not the current directory, you can enter a pathname like

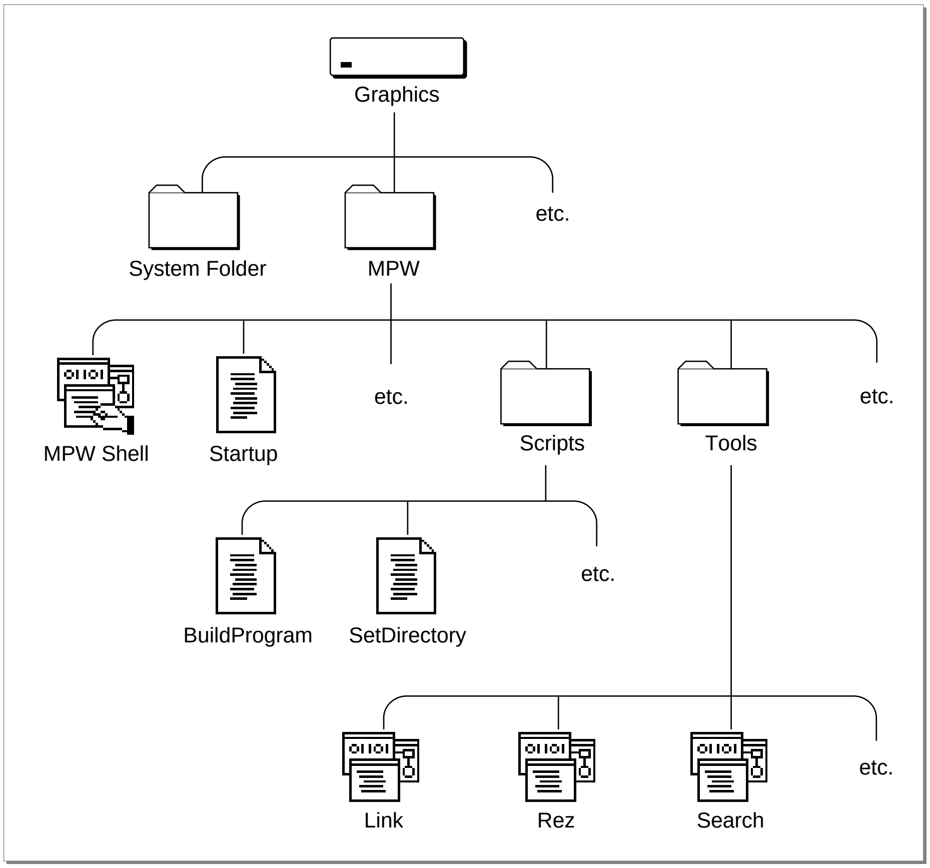
```
Graphics:MPW:ResEdit
```

and execute the pathname as if it were a command. Opening applications by entering the application's name as a command is useful when you want to open an application without leaving MPW.

Tutorial—Writing Pathnames

In this tutorial, you will practice writing pathnames for files. To complete this tutorial, you will use Figure 4-25, which is the same as Figure 4-20, but is repeated here for your convenience.

Figure 4-25 A sample directory structure



Each step of this tutorial asks you to write a pathname based on Figure 4-25, and the answers are given with each step. In this tutorial, you will

- write a full pathname, with or without a variable
- write a partial pathname
- write a pathname that uses multiple colons

For this tutorial, assume that the current directory is MPW. You can change the current directory for any step. You can also use MPW Shell variables within pathnames, but you may need to refer to Table 3-3 and Table 3-4 for the names and definitions of the variables.

To practice writing pathnames, follow these steps:

1. **Write the full pathname that locates the application MPW Shell.**

   This step has several possible answers, which are listed below, with comments following each:

   ```
   'Graphics:MPW:MPW Shell'
   "Graphics:MPW:MPW Shell"
   ```

   You must enclose the pathname in quotation marks because MPW Shell is two words. However, you can use either single or double quotation marks.

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

   The variable `{Boot}` is defined as the name of the volume where the system software is stored. In this case, `{Boot}` is equivalent to the expression `Graphics:`. It's wise to enclose this pathname in double quotation marks because the value of `{Boot}` might contain spaces or special characters. Remember that there is no colon after `{Boot}`.

   ```
   "{MPW}MPW Shell"
   ```

   This is the shortest way of writing the pathname. The variable `{MPW}` is defined as the full pathname of the directory in which the MPW Shell is stored. In this case, `{MPW}` is equivalent to `Graphics:MPW:`.This pathname also requires double quotation marks, because it contains a variable.

2. **Write a pathname that locates the script named `CreateMake`.**

   The full pathname would be

   ```
   Graphics:MPW:Scripts:CreateMake
   ```

   Since the current directory is MPW, you can also write a partial pathname:

   ```
   :Scripts:CreateMake
   ```

   To make the pathname even shorter, you can change the current directory to Scripts. In that case, you can use either of these pathnames to locate the script:

   ```
   CreateMake
   :CreateMake
   ```

The first pathname, `CreateMake`, is called a leafname, because it contains no colons. The second pathname, `:CreateMake`, works equally well.

3. **Change the current directory to Tools. Then, write the shortest possible pathname that locates the BuildProgram script, which is in the Scripts folder.**

   ```
   ::Scripts:BuildProgram
   ```

   Because the current directory is Tools, you must type a double colon to move up one level in the directory structure. Then, you enter the directory name Scripts and the filename BuildProgram.

When you write MPW Shell pathnames, remember that they are similar to the pathnames used in other systems, with each part of the pathname separated from the next by a colon. You can make a pathname shorter by including a variable, or by setting the current directory and writing a partial pathname from the current directory to the file. You can also use double colons as a shortcut to specify the parent directory, or triple colons to specify the grandparent directory.

Listing the Contents of a Directory

To list the files and directories in a directory, you can use the `Files` command with or without special wildcard characters. (Of course, you can also double-click on a folder icon to see a list of its contents in the Finder. The advantage of using `Files` is that you can generate a partial list of files that matches a specific pattern.)

In general, the syntax of the `Files` command is

```
Files [directoryName … ] [fileName … ] [option …]
```

If you do not specify a directory name, `Files` lists the entire contents of the current directory. If you specify the option `-d`, `Files` lists only the directories within the current directory. For example, if the current directory is MPW, the command

```
Files -d
```

would produce a list something like this one, but longer:

```
:Examples:
:Scripts:
:Tools:
```

You can recognize the items in the list as directories because they begin and end with colons. The command

```
Files -f
```

lists the full pathnames of each item in the current directory, as in

```
Graphics:MPW:Examples:
Graphics:MPW:Scripts:
Graphics:MPW:Tools:
'Graphics:MPW:MPW Shell'
```

The last pathname in the list is placed within single quotation marks because it contains a space. Generating a file list with full pathnames can be useful when you want to use a file's pathname in a command. You can use `Files -f`, then cut the pathname you want and paste it into a command.

To display a file list in multicolumn format, enter `Files` with the option `-m` followed by a number that specifies the number of columns. For example, the command

```
Files -m 2
```

displays a file list in two columns, like this:

```
:Examples:          :Tools:
:Scripts:           'MPW Shell'
```

To display the file list in a long format, with detailed information about the files in the list, use `Files -l`, which displays a list something like this:

```
Name          Type   Crtr    Size    Flags            Last-Mod-Date        Creation-Date

Examples      Fldr    Fldr   1068K    avbstclInmwod  2/19/92  11:00 am   3/27/90 4:48 PM
Scripts       Fldr    Fldr     94K    avbstclInmwod  2/19/92  11:00 am   3/27/90 4:48 PM
Startup       TEXT    MPS       4K    avbstclInmwod  4/17/91  11:00 am   4/14/91 3:21 PM
Worksheet     TEXT    MPS      16K    avbstclInmwod  4/17/91  11:00 am   4/14/91 3:21 PM
```

The meanings of the items in the Flags column are listed in Table 4-4.

Table 4-4 Definitions of the flags in the file list

Flag	Meaning
`A`	The file is a Finder alias file.
`a`	The file is not a Finder alias file.
`V`	The file is invisible from the Finder and from standard file dialog boxes.
`v`	The file is visible from the Finder and from standard file dialog boxes.
`B`	The file contains a bundle resource.*
`b`	The file does not contain a bundle resource.
`S`	The file or directory can't be renamed, and its icon can't be changed.
`s`	The file or directory can be renamed, and its icon can be changed.
`T`	The file is a stationery pad.
`t`	The file is not a stationery pad.
`C`	The file or directory has a customized icon.
`c`	The file or directory does not have a customized icon.
`L`	The file is write locked.
`l`	The file is not write locked.
`I`	The Finder has recorded information from the file's bundle resource and given the file or directory a position on the desktop.
`i`	The file does not yet have a position on the desktop.
`N`	The file contains no `'INIT'` resources.
`n`	The file contains `'INIT'` resources.
`M`	The file or application is available to multiple users.
`m`	The file or application is not available to multiple users.

Table 4-4 Definitions of the flags in the file list

Flag	Meaning
`W`	The file requires a switch launch.†
`w`	The file does not require a switch launch.†
`O`	The file is open.
`o`	The file is not open.
`D`	The file is on the desktop.†
`d`	The file is not on the desktop.†

* A bundle resource is a resource of type `'BNDL'` that the Finder uses to associate a file with its icon.
† System 7 does not use this flag.

By using the ? character and the ≈ character (Option-X) as wildcard characters with the `Files` command, you can write patterns that match a partial list of files. The ? character matches any single character, except a colon, a return, or a question mark. For example, the command

```
Files Sample.?
```

would match files with names like these:

```
Sample.c
Sample.a
Sample.h
```

If a filename contains a question mark, you would use `∂?` in the pattern you specify in your command because the ∂ character cancels the pattern-matching effect of the ? character. For example, you can enter

```
Files Sample.∂?
```

to match the file named Sample.?.

The ≈ character (Option-X) matches any string of characters, so the command

```
Files Sample.≈
```

matches a longer list of files, like this one:

```
Sample.c
Sample.a
Sample.h
Sample.make
Sample.incl.a
```

Note that if there is a directory that matches the pattern you specify with the `Files` command, all of the files in that directory will be listed.

The ? and ≈ characters are only two of the special characters that you can use as wildcard characters with the `Files` command. You will find a list of all of the MPW Shell special characters in the appendix "Table of Special Characters and Operators."

CHAPTER 5

Editing With Commands

Contents

You have already learned how to use the MPW Shell's editing features through the familiar Macintosh user interface—by choosing commands from menus, for example, or by entering text in dialog boxes. You can also use any of the MPW Shell's editing features by entering editing commands.

In fact, editing with commands gives you much more power and flexibility. For example, you can perform powerful find-and-replace functions, or you can combine commands into scripts to automate large editing operations.

This chapter explains how to use MPW Shell commands to edit text. Specifically, this chapter describes

- the commands and special characters used for editing
- the types of expressions used with editing commands
- techniques for writing correct expressions
- what to do when an editing command does not work

Using editing commands is an advanced skill that not all MPW developers need to acquire. Learning the material in this chapter is optional. However, since commands that perform editing operations are often used in scripts, you may wish to read this chapter and work through its tutorial before you read the chapter "Writing Scripts."

Using Special Expressions in Editing Commands

To write MPW Shell editing commands, you use two types of expressions known as regular expressions and selection expressions. A **regular expression** is a combination of text and special characters that specifies a pattern. Rather than specifying a specific text string, a regular expression might have many matches.

A **selection expression** is a formula that either selects a series of characters or positions the insertion point for the next editing command. A selection expression can include a specific number, a specific text string, or a regular expression. In other words, you can use a selection expression to search for a specific location, a word or phrase, or a string that matches a pattern.

The series of characters that a selection expression selects and highlights is called a **selection.** A selection is either a range of characters, such as a word or phrase, or the insertion point. The insertion point falls between two characters and has a range of zero characters.

You usually enter editing commands in the active window, and the commands act on the file in the target window unless you specify a window name or filename, as described in the next section.

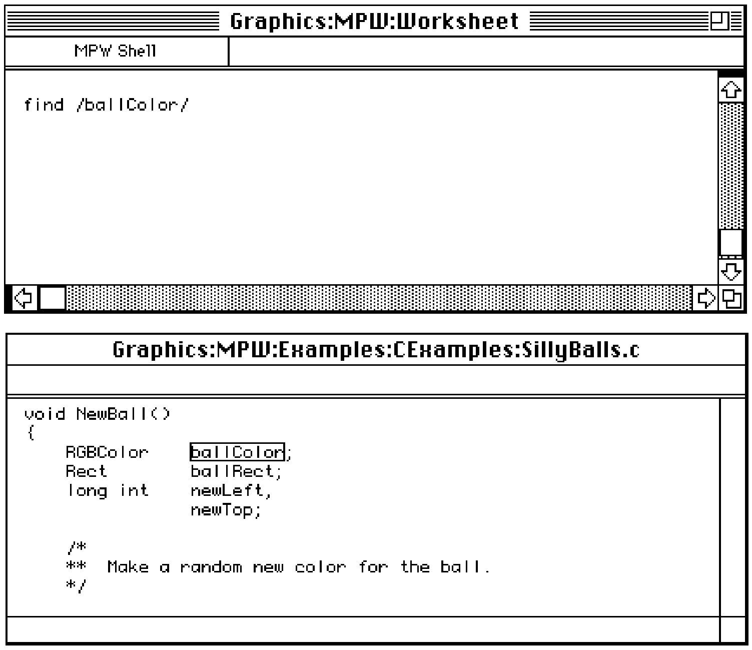
Using Target Windows

When you click in a window, it moves to the front and becomes the active window. All other windows on your screen except one are known as inactive windows.

The one window that is an exception is known as the target window. The target window is the window that was last active. The target window is unique because some commands you enter in the active window take effect there.

The target window is used primarily in text editing. For example, you can enter a text- editing command in the Worksheet window that acts on a source file in the target window. Notice that in Figure 5-1, the Worksheet window is the active window and the SillyBalls.c window is the target window.

Figure 5-1 Entering a command that acts on the target window

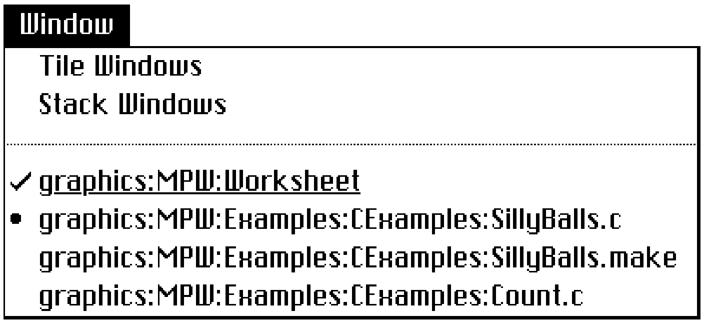


```
find /ballColor/
```

```
void NewBall()
{
    RGBColor    ballColor;
    Rect        ballRect;
    long int    newLeft,
                newTop;

    /*
    **  Make a random new color for the ball.
    */
```

The `Find` command in the Worksheet window has selected the string `ballColor` in the target window. When you have many windows open, you can tell which are the active and target windows by checking the Window menu (Figure 5-2).

Figure 5-2 Using the Window menu to check active and target windows


Window
Tile Windows
Stack Windows
✓ graphics:MPW:Worksheet
• graphics:MPW:Examples:CExamples:SillyBalls.c
graphics:MPW:Examples:CExamples:SillyBalls.make
graphics:MPW:Examples:CExamples:Count.c

The Window menu lists the titles of all the windows that are displayed, showing the full name for each window. The active window is marked with a √ character, and the target window is marked with a • character. In the Window menu in Figure 5-2, the Worksheet window is the active window and the SillyBalls.c window is the target window.

In the Window menu, a window name is underlined if you have made changes to that file but have not yet saved the changes. In Figure 5-2, you can see that the Worksheet window has changes that have not yet been saved.

Writing Simple Selection Expressions

A selection expression can locate

- the beginning or end of a file
- the beginning or end of a line
- a specific line, by its line number
- a specific line, by its distance forward or back from the insertion point or the current selection
- a series of characters
- the position just before or after a selection
- the position that is a certain number of characters before or after a selection
- two selections and everything in between

- the text that has been selected with the mouse

For example, each of the following commands includes a selection expression:

```
Find 3
Find !3
Find /shazam/
```

Note
The `Find` command is used throughout this chapter to illustrate the use of selection expressions and regular expressions. You can, however, use selection expressions with many other MPW Shell commands. To check whether a command uses selection expressions, you can use the online help or refer to *MPW Command Reference*. ◆

Moving to the Beginning or End of a File

To move the insertion point to the beginning of a file, you use a • character (Option-8) by itself in a command, as in

```
Find •
```

To move the insertion point to the end of a file, you use the ∞ character (Option-5) in the same way:

```
Find ∞
```

Moving by Lines

To move to a specific line in a file, you enter the `Find` command with a line number, as in

```
Find 102
```

which locates line 102 in the target window and selects the entire line. To select the line that is *n* lines forward from the insertion point or the current selection, use a ! character followed by a number in your command. For example,

```
Find !5
```

selects the line that is 5 lines forward from the insertion point or the current selection.

To select the line that is *n* lines back from the insertion point or the current selection, use a ¡ character (Option-1) followed by a number, as in

```
Find ¡5
```

which selects the line that is 5 lines back.

Note
When you enter a command to move forward and select text, the move forward starts from the insertion point or the *end* of the current selection. When you enter a command to move backward and select text, the move backward starts from the insertion point or the *beginning* of the current selection. ◆

Moving by Lines Plus Characters

To move the insertion point forward by a number of lines or a number of characters from the insertion point or the current selection, you would use the !*n* notation twice. For example, the command

```
Find !3!2
```

moves the insertion point 3 lines forward, selects the entire line, and then moves the insertion point 2 characters forward from the beginning of the next line. For example, you can enter the lines shown in Listing 5-1 in a file.

Listing 5-1 Some sample text

```
line 0
line 1
line 2
line 3
line 4
```

If you place the insertion point at the beginning of the lines, and then enter the command

```
Find !3!2
```

the insertion point is placed after the *i* in line 4. Now, if you enter the commands

```
Find •
Find !8
```

the MPW Shell selects the last line of the file, line 4.

To move the insertion point backward by lines plus characters, you use the ¡ character (Option-1) followed by a number of lines or a number of characters. The first ¡*n* notation in a command specifies the number of lines, and the second specifies the number of characters.

For example, if you place the insertion point at the end of the lines in Listing 5-1, and then enter

```
Find ¡3¡2
```

the insertion point is placed before the `0` in line 0. First, the line 3 lines back (which is line 1) is selected. Then, the insertion point moves 2 characters back, past the return character at the end of line 0, then past the `0` in line 0. Notice that the two commands

```
Find !3!2
Find ¡3¡2
```

are not exact opposites. That is, if you place the insertion point at the beginning of a set of lines, then enter both commands, the insertion point does not return to its starting point. (Remember that when you use the ! character to move forward, lines and characters are counted from the insertion point or the *end* of the current selection. When you use the ¡ character [Option-1] to move backward, lines and characters are counted from the insertion point or the *beginning* of the current selection.)

You can also combine the ! and ¡ characters in a command. If you enter a command like

```
Find !4¡7
```

the insertion point moves 4 lines forward, selects the line, then moves 7 characters back in the previous line. Likewise, if you enter

```
Find ¡2!5
```

the insertion point moves two lines back, selects the line, then moves 5 characters forward in the following line.

Specifying a Character or a String

Characters and text strings are among the easiest objects to locate. To search forward from the insertion point for a string, enclose the string in forward slashes, as in

```
Find /hello/
```

The command searches from the insertion point to the end of the file, and then stops, unless you have set the value of the variable `{SearchWrap}` to 1, meaning *on*. (You can use the `UserVariables` Commando dialog box to set `{SearchWrap}`, as explained in the chapter "Working With Files and Directories.")

To search backward for a string, enclose the string in backward slashes, as in

```
Find \goodbye\
```

When you place a string containing spaces or special characters within forward or backward slashes, you do not need to use quotation marks around the string. For example, the command

```
Find /Hello world/
```

does not need quotation marks, but locates the string

```
Hello world
```

even though the string contains a space. The string you place within forward or backward slashes does not need to be a full word. You can also use a regular expression to specify a pattern within forward or backward slashes, as explained in "Writing Selection Expressions That Contain Patterns" in this chapter.

Some MPW Shell commands include the option `-c` *n*, which lets you find the *n*th occurrence of a string. For example,

```
Find /hello/ -c 2
```

skips the first occurrence of the string `hello` and selects the second.

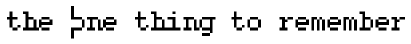
Positioning the Insertion Point Before or After a Selection

You can also place the insertion point just before or after a selection, or a certain number of characters before or after a selection. To place the insertion point at the beginning of a selection, you would type the Δ character (Option-J) before the slashes in your selection expression. For example, if you enter

```
Find Δ/one/
```

the insertion point would move just before the string `one`, as shown in Figure 5-3.

Figure 5-3 Placing the insertion point just before a selection

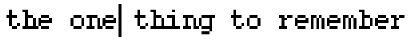


To place the insertion point just after the selection, you type the Δ character (Option-J) just after the slashes in your selection expression. For example, the command

```
Find /one/Δ
```

places the insertion point just after the string `one`, as shown in the example in Figure 5-4.

Figure 5-4 Placing the insertion point just after a selection

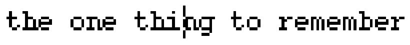


To place the insertion point a certain number of characters after a selection, you would use the notation !*n*; to place it before a selection, you would use ¡*n*. To add either notation to a selection expression, place it after the forward or backward slashes. For example, the command

```
Find /one/!4
```

places the insertion point 4 characters after the end of `one`, as shown in Figure 5-5 on page 6-10.

Figure 5-5 Placing the insertion point four characters after the selection

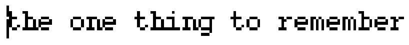

the one thing to remember

Likewise, the command

```
Find /one/¡4
```

places the insertion point four characters before the beginning of `one`, as shown in Figure 5-6.

Figure 5-6 Placing the insertion point four characters before the selection


the one thing to remember

Specifying Two Selections and Everything in Between

To specify two selections and everything in between them (for example, from one word to another and all the text in between), you enter two selection expressions with a : character between them. For example, the command

```
Find /main/:/end/
```

selects the first occurrence of `main`, the first occurrence of `end`, and all the text in between. Another useful example is the command

```
Find •:∞
```

which selects the entire contents of the file from top to bottom.

Specifying the Current Selection

In an MPW Shell command, the § character (Option-6) represents the current selection. In order to use the § character in a command, you must first open the target file and select some text. Once you do so, the command

```
Copy §
```

would copy the current selection to the Clipboard.

To move the insertion point to the beginning of the current selection, you would enter

```
Find Δ§
```

To move the insertion point to the end of the current selection, you would enter

```
Find §Δ
```

You can also use the § character in more complex commands. For example, the command

```
Replace § 'some text'
```

replaces the current selection with the string `some text`. If the current selection is an insertion point, the command inserts the replacement text at the insertion point, moving the insertion point to the end of the newly added text.

IMPORTANT
You cannot undo the result of a selection expression. If you are unsure of what an editing command might do, you may want to make a copy of the file you are working on and practice on it before you make changes to your file. ▲

With any command that takes a filename as a parameter, you can add the § character to the filename to have the command act only on the text that is selected in the file, rather than on the entire file. To add the § character to a filename, use the form *filename.*§. For example, the command

```
Print Sample.§
```

prints only the selected text in Sample, not the entire file. Likewise, the command

```
Compare Sample1.§ Sample2.§
```

compares the selected text in Sample1 with the selected text in Sample2.

If a command does not take a filename as a parameter, you cannot use the *filename*`.§` syntax. For example, the command

```
Set Text Sample.§     # this doesn't work
```

would set the value of `{Text}` to the string `Sample.§`, not to the selected text in the file Sample. However, to use selected text as a parameter when a command does not take filenames as parameters, you can use the `Catenate` command. For example, the command

```
Set Text "`Catenate Sample.§`"
```

sets the value of the variable `{Text}` to the currently selected text in Sample. You could then write a second command that acts on `{Text}`.

You can also use the notation *filename.*`§` as input to a command. For example, if you have defined a variable named `{Window}` as the name of the file in the target window, you could enter a command like

```
Translate a-z A-Z < {Window}.§
```

to convert all of the selected text in the target window from lowercase to uppercase. The output would be displayed in the active window, usually the Worksheet window. You would probably want to redirect the output to another file with a command similar to this one:

```
Translate a-z A-Z < {Window}.§ > Sample1
```

The new file, Sample1, would be stored in the current directory.

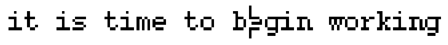
Grouping Expressions

When you write a long selection expression, you can use parentheses to group part of it. Expressions within parentheses have a higher precedence that those that are not; that is, expressions within parentheses are evaluated before those that are not. If a command contains several expressions within parentheses, the expressions are evaluated in order from left to right. For example, the command

```
Find (Δ/begin/)!1
```

would place the insertion point just before `begin`, then move it 1 character forward, as shown in Figure 5-7.

Figure 5-7 Grouping expressions to place the insertion point within a word



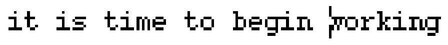
```
it is time to begin working
```

If you enter the same command with the parentheses placed differently, you would get a very different result. For example, the command

```
Find Δ(/begin/!1)
```

would place the insertion point 1 character past the end of `begin`, as shown in Figure 5-8.

Figure 5-8 Grouping expressions to place the insertion point after a word



```
it is time to begin working
```

In this case, the Δ character (Option-J) does nothing because it has no selection before which to move the insertion point. If expressions within parentheses are nested, the innermost expression is evaluated first.

Writing Selection Expressions That Contain Patterns

Now that you are familiar with writing simple selection expressions, you can move on to writing selection expressions that contain regular expressions. Regular expressions define patterns that can match any of a set of items and are a bit more complex than simple selection expressions.

You can use regular expressions to

- specify a pattern that occurs within a file
- specify a pattern that occurs at the beginning or end of a line
- write a list of characters so that any character in the list (or not in the list) is matched
- match a pattern that occurs a certain number of times
- tag a selection with a number so that you can refer to the selection by its number

The following sections explain each of the points listed above.

Specifying a Text Pattern With Wildcards

Regular expressions can include special pattern-matching characters, also known as wildcard characters. A **wildcard** is a character that can match any other character or any string. You can combine wildcard characters with literal characters to create a pattern.

To match any single character, you would use the `?` character, as in

```
Find /what the ?/
```

which could match many strings, including these:

```
what the h
What the 2
```

To match a string of characters, including a single character or the null string, you would use the ≈ character (Option-X). For example, the command

```
Find /far≈/
```

could match many strings, including these:

```
farm
farther
farout
far
```

To use a wildcard character as a literal character, precede it with a ∂ character (Option-D) or enclose it in single or double quotation marks. For example, any of the commands

```
Find /What the ∂?/
Find /What the '?'/
Find /What the "?"/
Find /'What the ?'/
```

would cancel the pattern-matching effect of the `?` character and match only

```
What the ?
```

because the `?` character is taken literally.

Specifying a Text Pattern at the Beginning or End of a Line

To match a pattern only when it occurs at the beginning of a line, you would use the • character (Option-8) just before the pattern, placing it within the slashes. For example, the command

```
Find /•main/
```

matches the string `main` only if it appears at the beginning of a line.

To match a pattern only when it occurs at the end of a line, you would use the ∞ character (Option-5) just after the pattern, also placing it within the slashes. For example, the command

```
Find /end∞/
```

matches `end` only when it occurs at the end of a line. The end of a line is either the last character of a line before the return, or the end of a file. In the command

```
Echo This is really one ∂
 line, not two
```

which is spread across two physical lines, the end of the first line is at the ∂ character. If you entered the command

```
Find /one∞/     # this doesn't work
```

the MPW Shell would not select the string `one` because it is not at the end of the line. However, the command

```
Find /∂∂∞/      # but this does
```

would select the ∂ character at the end of the first line. In the command, the first ∂ character indicates that the second ∂ character should be taken literally. Likewise, the command

```
Find /one ∂∂∞/
```

selects the word `one`, the space, and the ∂ character at the end of the line.

Using Scan Sets

A **scan set** is a special type of regular expression consisting of one or more characters within brackets (that is, within `[` and `]` characters). You can write a scan set so that it matches any character in a list, any character not in a list, or any of a range of characters.

Once you write a scan set, you can place it within forward or backward slashes for a forward or backward search. For example, you can enter

```
Find /[ABCDE]/
```

to match any of the characters `A`, `B`, `C`, `D`, or `E` in uppercase or lowercase. Note that scan sets are not case sensitive, unless you either set the variable `{CaseSensitive}` to 1 or enter the scan set in the Find dialog box and click the Case Sensitive checkbox.

To make writing scan sets easier, you can use two characters separated by a hyphen to specify an inclusive range, as in

```
Find /[A-E]/
```

which also matches any of the characters `A`, `B`, `C`, `D`, or `E` in uppercase or lowercase. These commands all specify inclusive ranges:

```
Find /[a-z]/
Find /[0-9]/
Find /[A-F0-9$]/
```

The last command in the list matches any of the characters `A` through `F`, any of the numerals `0` through `9`, or the `$` character.

If you make the ¬ character (Option-L) the first character in the set, the command matches any character *not* in the set, so that

```
Find /[¬ABCDE]/
```

would match any character other than uppercase or lowercase `A`, `B`, `C`, `D`, or `E`.

To specify any of the special characters

```
∂  /  \  '  "  [  ]
```

as literal characters in a scan set, you would place a ∂ character just before the special character. For example, the command

```
Find /[a-z∂/]/
```

matches any letter or the `/` character. To specify the `[` character in a scan set, you would enter

```
Find /[∂[]/
```

Likewise, to specify the `]` character in a scan set, you could use the command

```
Find /[∂]]/
```

To specify a hyphen as a literal character in a scan set, you would place it first in the set, like this:

```
Find /[-a-f]/
```

However, to specify the ¬ character (Option-L) as a literal character in a scan set, place it anywhere in the list except first, as in this command:

```
Find /[a-f¬]/
```

When you write scan sets, remember to use a `]` character to close a `[` character, and to place the scan set within forward or backward slashes.

Repeating Regular Expressions

You can add a special character to a regular expression to specify that a match can occur one or more times in succession. This is one of the techniques that make regular expressions very powerful. For example, as described in the last section, the command

```
Find /[a-z]/
```

finds any lowercase letter between `a` and `z`. When you add the + character, which specifies one or more occurrences, as in

```
Find /[a-z]+/
```

the command finds any sequence of lowercase letters between `a` and `z`. Likewise, you could replace a string of one or more tabs with one tab by entering

```
Replace /∂t+/ ∂t
```

To specify that a match for a regular expression can occur zero or more times in succession, you would use the `*` character after the scan set. For example,

```
Find /and[ ]*/
```

finds the string `and` when it is followed by zero or more spaces. In other words, it finds `and` when it is a whole word or part of a word. Note that you do not need to place quotation marks around the space that is placed inside brackets.

To specify that a match for a regular expression should occur *n* times in succession before it is matched, place the number *n* within « and » characters (Option-backslash and Option-Shift-backslash) after the regular expression. For example, if you enter

```
Find /[ ]«5»/
```

the MPW Shell finds a string of exactly 5 spaces.

There are two variations on the use of the « » characters. The notation «*n*,» specifies that a match can occur at least *n* times in succession, and the notation «n_1,n_2» specifies that a match should occur at least n_1 times and at most n_2 times in succession.

For example, the command

```
Find /[∂n]«1»/
```

finds exactly one ∂n, or return character. However, the command

```
Find /[∂n]«1,»/
```

matches any sequence of one or more return characters. The command

```
Find /[∂n]«1,4»/
```

finds a sequence of one, two, three, or four return characters.

When you specify a number of repetitions with any of the notations `+`, `*`, «*n*», «*n*,», or «n_1,n_2», be sure to place the notation inside the slashes that enclose the selection expression.

Tagging Selection Expressions

Sometimes it is useful to know what selection matched a certain part of a selection expression. For example, you might want to know the string that matched so that you can use it in a different parameter of the same command.

To tag part of a selection expression, enclose that part in parentheses and then add the ® character (Option-R) followed by a number between 0 and 9. For example, you can enter

```
Find /([0-9])®1/
```

to search for any of the numerals between 0 and 9, tagging the text that is selected with the number 1. Be certain to place the ®*n* notation inside the slashes.

Once you select some text and then tag it, you can refer to it later in the same command. For example, you can enter

```
Replace /([0-9]+)®1,([0-9]+)®2/ "®2,®1"
```

to reverse two numerals separated by a comma; for example, to change `1,3` to `3,1`. The command has two parameters. The first is a pattern that `Replace` searches for, and the second is the string that replaces it.

To analyze the command, break it down into parts, as follows:

Part of the command	Meaning
`/([0-9]+)®1,([0-9]+)®2/`	This entire parameter is within forward slashes to indicate a forward search.
`([0-9]+)®1`	The range `[0-9]` specifies the range of numbers between 0 and 9. The `+` character specifies that the string can contain one or more numerals between 0 and 9. The parentheses group part of the expression, and the `®1` tags it with the number 1.
`,`	The comma is a literal character that specifies that a comma should occur between the two strings.
`([0-9]+)®2`	This part works the same way as the part tagged `®1`.
`"®2,®1"`	This parameter reverses the order of the two numbers the first parameter selected. The double quotation marks around this parameter are optional.

Another useful example is selecting all the volume names in a list of pathnames. For example, you might want to change the pathnames

```
Graphics:MPW:Tools:Print
Graphics:MPW:Examples:CExamples:SillyBalls.c
```

to

```
NewVol:MPW:Tools:Print
NewVol:MPW:Examples:CExamples:SillyBalls.c
```

To make this change, you would use this command:

```
Replace /[¬:]+:(≈)®1/ "NewVol:®1"
```

Even though the command is rather complex, it is fairly easy to understand if you break it down into parts. The parts of the command and their meanings are as follows:

Part of the command	Meaning
`/[¬:]+:(≈)®1/`	This entire parameter, which specifies the volume name, is within forward slashes to indicate a forward search.
`[¬:]+`	The range `[¬:]+` is matched by any string of characters that does not contain a colon.
`:`	The colon is a literal character that specifies that the string should be followed by a colon.
`(≈)®1`	The ≈ character specifies that the colon is followed by any string of characters. The parentheses allow the string to be tagged. The notation `®1` tags the string. Note that the parentheses do not cause this expression to be evaluated first.
`"NewVol:®1"`	This entire parameter is enclosed in double quotation marks so that the ® character is interpreted as a special character. This parameter replaces the old volume name with the string `NewVol:`, followed by the string that the `(≈)®1` selected.

The ® character behaves differently in different commands. Some commands create a variable named `{®`*n*`}` whose value is the string selected by ®*n*. You can then use the variable `{®`*n*`}` in subsequent commands. In other commands, you can use the selection that matches the ®*n* notation only in the same command; and in still other commands, the ® character is ignored.

The `Evaluate` command is one of the commands that creates a variable named `{®`*n*`}`. For example, the command

```
Evaluate "{MPW}" =~ /([¬:]+:)®1≈/
```

expands the variable `{MPW}` to the volume name on which the MPW software is stored, and then stores the volume name in a variable named `{®1}`. You can now use `{®1}` in subsequent commands. For example, you can enter

```
Echo "Your MPW directory is on the volume {®1}"
```

and the MPW Shell might respond with

```
Your MPW directory is on the volume Graphics:
```

The ways in which MPW Shell editing commands handle the ® character are summarized in Table 5-1. You can find another example of how to use the ® character in the following tutorial.

Table 5-1 How MPW Shell commands handle the ® character

Command	**How it uses ®**
`Adjust`	Ignores ®
`Align`	Ignores ®
`Break If`	Creates the variable `{®`*n*`}`
`Clear`	Ignores ®
`Continue If`	Creates the variable `{®`*n*`}`
`Copy`	Ignores ®
`Cut`	Ignores ®
`Evaluate`	Creates the variable `{®`*n*`}`
`Exit If`	Creates the variable `{®`*n*`}`
`Find`	Ignores ®
`If`	Creates the variable `{®`*n*`}`
`Line`	Ignores ®
`Mark`	Ignores ®
`Paste`	Ignores ®
`Replace`	As ®*n*
`Search`	Ignores ®
`StreamEdit`	As ®*n*

Tutorial—Reversing the Order of Columns in a File

For this tutorial, you will create a table consisting of two columns of numbers that look like this:

```
123            456
123            456
123            456
123            456
```

Suppose that the columns are separated by one or more tab characters and you want to switch the order of the columns. In this tutorial, you will build a command that can do so. Later, you can use this type of powerful editing command to automate the editing of long or complex files.

You will build the command step by step. To build the command, you will

- write a regular expression that matches any line in the table
- tag each of the columns the regular expression matches
- find the first line that matches the regular expression
- reverse the order of the tags, thereby reversing the columns
- repeat the operation for each line in the file

To see how each general step works, follow the detailed instructions in these steps:

1. **Open a new file in which to enter the sample table.**

   You can name the new file Table, or whatever you like.

2. **Type the following table, adding one tab character between the two columns and pressing Return at the end of each line:**

   ```
   123         456
   123         456
   123         456
   123         456
   ```

3. **Place the insertion point at the beginning of the table.**

4. **Type this regular expression in the Worksheet window:**

   ```
   [0-9]+
   ```

   This regular expression matches any group of one or more numbers.

5. **Enclose the regular expression in parentheses, then type ®1 to tag the text the expression selects:**

   ```
   ([0-9]+)®1
   ```

6. **Type `[∂t]+` to add at least one tab character to the regular expression to allow for the space between the columns:**

```
([0-9]+)®1[∂t]+
```

7. **To match the second column and tag the selected text with `®2`, type**

```
([0-9]+)®2
```

   The regular expression now looks like this:

```
([0-9]+)®1[∂t]+([0-9]+)®2
```

8. **Place forward slashes around the regular expression to make it a selection expression:**

```
/([0-9]+)®1[∂t]+([0-9]+)®2/
```

9. **To switch the order of the columns, type a second regular expression:**

```
"®2∂t®1"
```

   This regular expression displays the ®2 text, followed by a tab character, followed by the ®1 text.

10. **Put both expressions into a `Replace` command:**

```
Replace -c ∞  /([0-9]+)®1[ ∂t]+([0-9]+)®2/  "®2∂t®1"
```

   The option `-c ∞` specifies that the operation should be repeated as many times as possible until the end of the file is reached.

11. **Place the insertion point on the same line as the `Replace` command, then press Enter.**

   After you enter the command, the table looks like this:

```
456          123
456          123
456          123
456          123
```

As you can see from the example, editing commands can be quite powerful and can save hours of manual labor. Although the table used in this example is short, the same process applies to much longer files. To write a complex regular expression, it is useful to break it into parts. Remember that you can refer to the information in this chapter or to the appendix “Table of Special Characters and Operators” for help.

Summary of Selection Expressions

The types of selection expressions you can use in MPW Shell commands are summarized in Table 5-2.

Table 5-2 The types of selection expressions

Expression	Meaning
•	The beginning of the file
∞	The end of the file
n	A line number
!*n*	*n* lines forward
¡*n*	*n* lines backward
!n_1!n_2	n_1 lines plus n_2 characters forward
¡n_1¡n_2	n_1 lines plus n_2 characters backward
!n_1¡n_2	n_1 lines forward, then n_2 characters backward
¡n_1!n_2	n_1 lines backward, then n_2 characters forward
Δ*expression*	The position before the first character of the selection
*expression*Δ	The position after the last character of the selection
expression!*n*	*n* characters forward from the end of the selection
expression¡*n*	*n* characters back from the beginning of the selection
expression : *expression*	Both selections and everything in between
§	The current selection
(*expression*)	A selection grouping
/ *regularExpression* /	Searching forward, the first string that matches the pattern
\ *regularExpression* \	Searching backward, the first string that matches the pattern

How Selection Expressions Are Evaluated

Selection expressions are evaluated according to the precedence of the special characters they include. Those selection expressions that contain characters of higher precedence are evaluated before selection expressions with characters of lower precedence.

Table 5-3 lists the special characters your selection expression might contain in order of decreasing precedence. When two characters are listed for one precedence number, both characters have equal precedence.

Table 5-3 The selection characters, from highest precedence to lowest

Precedence	Character	Meaning
1st	/ /	Searches forward
	\ \	Searches backward
2nd	( )	Evaluates first
3rd	Δ	Positions the insertion point
4th	!	Moves forward
	¡	Moves backward
5th	:	Joins two selections

Summary of Regular Expressions

The types of regular expressions are summarized in Table 5-4. Remember that regular expressions are used as selection expressions.

Table 5-4 The types of regular expressions

Expression	Meaning
?	Any character except return
≈	Any string of characters, including the null string
∂*s*	A special character used as a literal character
•*expression*	A pattern when it occurs at the beginning of a line

Table 5-4 The types of regular expressions (continued)

Expression	Meaning
expression∞	A pattern when it occurs at the end of a line
[*scanSet*]	Any character in the scan set
[¬ *scanSet*]	Any character not in the scan set
[c_1-c_2]	Any character between c_1 and c_2, inclusive
expression+	The expression when it occurs one or more times in succession
*expression**	The expression when it occurs zero or more times in succession
expression«*n*»	The expression when it occurs exactly *n* times in succession
expression«*n*,»	The expression when it occurs at least *n* times in succession
expression«n_1,n_2»	The expression when it occurs at least n_1 times and at most n_2 times in succession
(*expression*)®*n*	A tag indicating that you can refer to the text that matches the regular expression as ®*n* (or in some commands, as the variable{®n})

Solving Expression Difficulties

When a selection expression or regular expression does not select or match what you intended, you may find the following:

- Special characters are not properly quoted.

For example, the `(` character is a special character. To search for it, you must either precede it with the ∂ character or enclose it within quotation marks, as in:

```
∂(
'('
```

You must also quote other special characters in the same way.

- Characters that must be used in pairs might be missing one of the pair.

  Many characters, such as these, must be used in pairs:

```
/   /
(   )
"   "
'   '
\   \
[   ]
{   }
```

  If you use one character of the pair without its mate, the MPW Shell generates an error message.

- Special characters are not used correctly. You can refer to the description of any special character in the appendix “Table of Special Characters and Operators.”
- The precedence and usage of characters in your selection expression are not correct.

  For example, these two `Find` commands have slightly different syntax and very different meanings:

```
Find •:/main/
Find /•main/
```

  The first command selects everything from the beginning of the file until the first occurrence of `main`. The second command matches the string `main` when it occurs at the beginning of a line. To determine the precedence of the characters you use in regular and selection expressions, you can consult Table 5-3 on page 6-23.

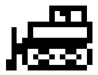
- The individual pieces of the selection expression do not select what you intended.

Break a difficult selection expression into small parts. Try each part separately to make sure that it does what you want. Then add each new, tested part individually to create more complex selection expressions.

Handling a Memory Problem During Editing

If you are making a very large number of editing changes, the memory in the MPW Shell may become fragmented. If so, the MPW Shell might have trouble finding a contiguous block of memory to use in order to make the changes. If that happens, the MPW Shell displays the bulldozer cursor (Figure 5-9).

Figure 5-9 The bulldozer cursor, at twice its normal size



The bulldozer tells you that the MPW Shell is trying to clear more memory so that it can process your file. If the usual cursor reappears, you can save the file that the editing commands are working on, thus reinitializing the file's memory area.

To avoid problems caused by low memory, you should try to write your command (or commands) so that it proceeds in stages and saves the file periodically. It is the unsaved additions to the file that cause the MPW Shell to run out of memory.

For example, suppose you have a file of 10,000 lines that you wish to edit with the commands shown in Listing 5-2.

Listing 5-2 Some commands that might cause problems

```
Clear -c ∞ /•  /
Replace -c ∞ /"("/ "["
```

In Listing 5-2, the `Clear` command searches for any lines that begin with two spaces and clears the spaces in each line. Then, the `Replace` command replaces all occurrences of the `(` character with the `[` character throughout the file.

These two commands would probably require a very large amount of memory. However, you can modify the commands so that after the first command, the file is saved and closed and then reopened, as shown in Listing 5-3.

Listing 5-3 Stopping after the first command is processed

```
Clear -c ∞ /•  /
Close -y "{MyEditWindow}"
Open -t "{MyEditWindow}"
Replace -c ∞ /"("/ "["
```

But what if the bulldozer still appears, coming during the processing of the first command? In that case, you would modify your commands to stop processing at regular intervals, as shown in Listing 5-4.

Listing 5-4 Stopping at regular intervals

```
Find •
    Loop
        Clear -c 4000 /•  /
        Break If {Status} != 0
        Close -y {MyEditWindow}
        Open -t {MyEditWindow}
    End
```

In these commands, the `Clear` command stops after the first 4000 lines, saves and closes the file, and then reopens the file. When the file is reopened, the insertion point remains where it was last placed. Then the editing process returns to the top of the file and continues for another 4000 operations. Note that the first command, `Find •`, is very important. If you leave it out, the operation does not work.

C H A P T E R 6

Writing Scripts

Contents

By now you know that the MPW Shell command language is, in fact, a small programming language that includes input and output redirection, piping, Shell variables, and selection expressions.

This chapter describes how to combine commands into scripts. You can use scripts to customize your environment, perform time-consuming tasks such as converting assembly language to hexadecimal code, and automate large editing operations, such as searching for and replacing a pattern in a large group of files.

This chapter introduces special command skills that you use primarily in scripting and then goes on to describe how to combine commands into scripts. Specifically, this chapter describes how you

- write the complex commands you need for scripts
- begin a script
- write commands to handle the parameters your user enters to start your script
- have your script display alert and dialog boxes
- handle errors in your scripts

The material presented in this chapter is somewhat advanced. You do not need to understand this chapter to develop software with MPW. However, if you want to develop scripts, be sure that you are familiar with the material in all of the preceding chapters, especially the material on entering commands and writing regular and selection expressions.

Your MPW software contains a folder of sample scripts named MPW Script Tips. This chapter contains excerpts from those scripts, to illustrate various scripting techniques. If you wish to see any script in its entirety, you can do so online.

Some Advanced Command Skills

The following sections describe the advanced skills you need to acquire before you can write scripts. In these sections, you learn how to

- write commands that compare values

- write control structures that specify the order in which commands are interpreted
- define variables that contain a list of items
- nest embedded commands
- prevent quotation marks from being discarded

These are command skills that you use primarily when writing scripts. After you read this section, you can turn to the section “Combining Commands Into Scripts” on page 7-27 to begin writing scripts.

Comparing Values in Commands

As you write scripts, you frequently use arithmetic and logical expressions to compare values in commands. Arithmetic and logical expressions are formulas that you can use in a command to define a calculation to be performed. Arithmetic and logical expressions can act on numbers, text strings, or variables. (In the rest of this section, arithmetic and logical expressions are called simply **expressions.**)

The values in the expression are known as **operands.** Operands are compared or combined by an **operator,** which is usually a special character or a combination of special characters. For example, in the expression

```
2 == 2
```

`2` and `2` are both operands, and the == character, which means *equal to*, is the operator. Ordinarily, expressions follow a command, as in

```
Evaluate 2 == 2
```

which evaluates whether `2` is equal to `2`. In this case, the result of the command is 1, meaning `TRUE`.

You could also enter

```
Evaluate {a} == 2
```

which evaluates whether the current value of the variable `{a}` is `2`.

The MPW Shell has a wide range of operators that you can use in expressions, and these operators are described in the following sections.

Comparing Text Strings, Numbers, and Variables

To write an expression comparing two values that are read as strings, such as a text string or a number, use the == character, which means *equal to*, or the != character, which means *not equal to*.

For example, if you enter

```
Evaluate Alpha == Alpha
```

the MPW Shell returns 1, meaning `TRUE`; and if you enter

```
Evaluate Alpha != Beta
```

the MPW Shell returns 1 again, or `TRUE`, because the != character tests for strings that are not equal.

With the == and != characters, a match is case sensitive, so that the command

```
Evaluate Alpha == alpha
```

returns 0, or `FALSE`. You do not need to use quotation marks around a text string used as an operand unless it contains spaces or special characters.

Note
Note that variable names are not case sensitive. In other words, the variables `{Alpha}` and `{alpha}` have the same value to the MPW Shell. ◆

You can also use the == and != characters to evaluate variables because variables contain text strings or numbers. For example, you can enter the commands

```
Set A Alpha
Set B Alpha
Evaluate {A} == {B}
```

to define the variables `{A}` and `{B}`and then compare their contents. Because `{A}` and `{B}` contain the same string, the result is 1, or `TRUE`. The same process works for these commands:

```
Set A 1
Set B 100
Evaluate {A} == {B}
```

In this case, the result is 0, or `FALSE`.

Comparing Regular Expressions to Text Strings

To write an expression comparing a text string with a regular expression that defines a pattern, you can use the `=~` character (*equal to, pattern*) or the `!~` character (*not equal to, pattern*). When you write the expression, you make the text string the left operand and the regular expression the right operand. You also place the right operand in forward slashes.

For example, the expression

```
Evaluate 'hello' =~ /hel≈/
```

tests whether the string `hello` matches the regular expression `/hel≈/`, and returns 1, or `TRUE`. The left operand can be a variable that contains a text string, as in

```
Evaluate "{Filename}" =~ /≈.c/
```

This command checks whether the current value of `{Filename}` ends with `.c`. Likewise, if you define a variable named `{bankAccount}` that contains an account number, the command

```
Evaluate {bankAccount} !~ /01-≈/
```

compares the value of `{bankAccount}` with the pattern `/01-≈/`, and returns the value 1 (`TRUE`) only if the value of `{bankAccount}` does not begin with `01-`.

Evaluating Arithmetic and Logical Expressions

When you write arithmetic and logical expressions, you can use any of the arithmetic or logical operators shown in Table 6-1.

Table 6-1 The arithmetic and logical operators

Operator	Keystroke	Meaning
`(  )`		Group expressions
`+`		Addition
`-`		Subtraction
`*`		Multiplication
`÷`	Option-slash	Division
`DIV`		Division
`%`		Modulus
`MOD`		Modulus
`==`		Equal to (strings, numbers, and variables)
`!=`		Not equal to
`<>`		Not equal to
`≠`	Option-=	Not equal to
`=~`		Equivalent to, patterns
`!~`		Not equivalent to, patterns
`>`		Greater than
`>=`		Greater than or equal to
`≥`	Option->	Greater than or equal to
`<`		Less than
`<=`		Less than or equal to
`≤`	Option-<	Less than or equal to
`&&`		Logical `AND`
`AND`		Logical `AND`
`!`		Logical `NOT`
`<>`		Logical `NOT`
`¬`	Option-L	Logical `NOT`
`||`		Logical `OR`
`OR`		Logical `OR`
`&`		Bitwise `AND`
`|`		Bitwise `OR`
`^`		Bitwise `XOR`

Table 6-1 The arithmetic and logical operators (continued)

Operator	Keystroke	Meaning
`<<`		Shift left
`>>`		Shift right
`-`		Subtraction, sign change
`~`		Bitwise `NOT`
`+=`		Increment a variable
`-=`		Decrement a variable

You can use any of the arithmetic operators in an arithmetic expression. The parts of an expression within parentheses are evaluated first. When an expression contains more than one group of parentheses, the groups are evaluated in order from left to right. For example, the command

```
Evaluate (4 * 6 * 7) ÷ (27 - 3)
```

yields

```
Evaluate 168 ÷ 24
```

the result of which is 7. You can also use the arithmetic operators with variables, as in

```
Evaluate {Value1} ≥ {Value2}
```

Note that all arithmetic expressions use integer arithmetic. For example, the expression 5 ÷ 2 yields 2, not 2.5.

The logical `AND` operators (`&&` or `AND`) evaluate a statement and return a value of 1, or `TRUE`, only if both sides of the statement are not equal to 0. If either side or both sides are equal to 0, the command returns 0, or `FALSE`. For example, the command

```
Evaluate 2 && 3
```

returns 1, or `TRUE`. If the command

```
Evaluate {Revenue} AND {Expenses}
```

returns 1 (`TRUE`), you know that neither the value of `{Revenue}` nor the value of `{Expenses}` is 0.

Note
The `&&` character has two meanings. When `&&` comes between two values, it means logical `AND`. When `&&` ends one command and introduces another, it means if the first command is successful, do the next command. It should be clear from the context of the command which meaning of `&&` is used. ◆

The logical `OR` characters (`||` or `OR`) also evaluate a statement, returning the value 1 (`TRUE`) if either side of the statement is not equal to 0 and the value 0 (`FALSE`) only if both sides are equal to 0. Therefore, the command

```
Evaluate 2 || 0
```

returns 1 (`TRUE`), as does the command

```
Evaluate {X} OR {Y}
```

if either `{X}` or `{Y}` is not equal to 0.

Note
The `||` character also has two meanings: when it comes between two values, it means logical `OR`, and when it ends one command and introduces another, it means that if the first command is successful, the second command should be executed. ◆

The Precedence of Operators

When you use more than one operator in an expression, the operators are evaluated in a certain order, as shown in Table 6-2. When more than one operator is shown within a group, each operator in the group has equal precedence.

Table 6-2 The expression operators, from highest to lowest precedence

Precedence	Operator	Meaning
1st	`( )`	Group commands or expressions
2nd	`-`	Subtraction, sign change
	`~`	Bitwise `NOT`
	`!`, ¬, or `NOT`	Logical `NOT`
3rd	`*`	Multiplication
	÷ or `DIV`	Division
	`%` or `MOD`	Modulus
4th	`+`	Addition
	`-`	Subtraction
5th	`<<`	Shift left
	`>>`	Shift right
6th	`<`	Less than
	`<=` or ≤	Less than or equal to
	`>`	Greater than
	`>=` or ≥	Greater than or equal to
7th	`==`	Equal to (strings, numbers, and variables)
	`!=`, `<>`, or ≠	Not equal to (strings, numbers, and variables)
	`=~`	Equivalent to (patterns)
	`!~`	Not equivalent to (patterns)
8th	`+=`	Increment a variable
	`-=`	Decrement a variable

Writing Control Structures

In general, MPW Shell commands are executed sequentially, one after the other. You can also create **control structures,** or groups of commands that are not executed sequentially, as commands ordinarily are, but that create conditional loops, structured loops, and iterations.

You can use control structures interactively by entering them in the Worksheet window, but usually you use them in scripts. Either way, you can nest one control structure within another as long as there is sufficient stack space. Control structures frequently use arithmetical and logical expressions, regular expressions, and selection expressions.

Once the MPW Shell executes the first command in a control structure, it does not execute any of the following commands until it finds the end of the control structure, usually an `End` command. While the MPW Shell reads the commands in the structure, the status panel displays the name of the first command.

Each of the types of control structures is described in the sections that follow.

The If … Else … End Structure

The `If … Else … End` structure contains expressions and commands and executes the list of commands that follow the first `TRUE` expression. The general form of the `If … Else … End` structure is

```
If expression1
    command1
    [ command2 … ]
[Else If expression2
    command3
    [ command4 … ] ]…
[Else expression3
    command5
    [ command6 … ] ]…
End
```

Each block of commands is either executed or skipped, depending on whether the expression just before it is `TRUE` or `FALSE`. Even though several expressions may be true, only one set of commands is executed. An example of the `If … Else … End` structure is shown in Listing 6-1.

Listing 6-1 Using the If …Else …End structure to generate a beep

```
If "{Status}" == 0
    Beep 1a,25,200
Else
    Beep -3a,25,200
End
```

The commands in Listing 6-1 check the value of the variable `{Status}`, which stores the result of the last command that was executed. If the value of `{Status}` is 0, showing that the last command was successful, `Beep` tells you with a high-toned beep. If `{Status}` has any other value, showing that the last command was not successful, `Beep` also tells you, this time with a low-toned beep.

To execute the commands in the structure, type them into the Worksheet window, select all of them, then press Enter. You will hear one beep or the other, depending on whether the last command you entered was successful.

To hear the "successful" beep, you can type a command like

```
Echo Hello
```

before you execute the lines in the listing. To hear the "unsuccessful" beep, type a command like

```
Echo "Hello
```

which causes `{Status}` to be set to –3; then execute the lines in the listing. You can also test the `If … Else … End` structure with the example given in Listing 6-2.

Listing 6-2 Another example of the If … Else … End structure

```
If "{Status}" == 0
    Echo Success!!
Else If "{Status}" == 1
    Echo Parameter Error
Else If "{Status}" == 2
    Echo Processing Error
Else
    Echo Unknown Error
End
```

Quite often, your script needs to test more than one condition in succession. You can do so by nesting `If … Else … End` structures, like this:

```
If expression
    command1
    [ command2 … ]
Else
    If expression
        command1
        [ command2 …  ]
    Else
        command1
        [ command2  …  ]
    End
End
```

The MPW Shell gives you a shorthand method for writing such structures. Instead of using the structure given above, you can enter

```
If expression
    statement  …
Else If expression
    statement …
Else If expression
    statement  …
Else
    statement  …
End
```

Notice that in the structure just given, the `Else If` and `Else` commands are not nested and that the structure has only one `End` command.

The For … End Structure

The `For … End` control structure repeats a set of commands for each parameter in a list. To define the list, you must enter the `For` command with a variable name, followed by the word `In` and a list of parameters. You do not need to define the variable before you use it in the `For` command; `For` defines it for you. On the next lines, you can specify commands to be carried out for each item in your `For` command. When each item in the list has been processed, `End` ends the structure.

The syntax of the `For … End` structure is as follows:

```
For name In parameter1 [parameter2] …
    command1 …
    command2 …
End
```

The commands you add between the `For` and `End` statements are executed once for each item in the parameter list. The current parameter is assigned to the variable `{`*name*`}`, and you can refer to `{`*name*`}` in the commands in the loop. An example of the `For … End` structure is shown in Listing 6-3.

Listing 6-3 Using the For … End structure to display a line number

```
For Line In 1 2 3
    Alert "The current line number is {Line}"
End
```

If you execute the lines shown in Listing 6-3, `Alert` displays an alert box for each line number. In this example, the items in the list are typed in the command, separated by spaces. If you were using this type of structure in a script, you would probably use an embedded command as a parameter to `Alert` to find the current line number.

Another example of using the `For … End` structure is shown in Listing 6-4.

Listing 6-4 Using the For … End structure to print each file in a list

```
For File in ≈.c
    Print -ps 'Sample PS File' "{File}"
    Echo "{File}" is printed.
End
```

The commands in Listing 6-5 print each file using the PostScript file named `Sample PS File`, and display a message in the active window as each file is printed. In the `For` command, the expression `≈.c` is used to create the list of files that the commands process.

The Loop … End Structure

The `Loop … End` structure creates a structured loop. That is, any commands between `Loop` and `End` are executed until the loop is terminated with a `Break` command. The `Break` command is required, and it terminates the loop if the condition defined by *expression* is met.

The syntax of the `Loop … End` structure is as follows:

```
Loop
    command1
    [command2] …
End
```

When the `Loop` structure terminates, the last command that was executed returns its status to `{Status}`. A simple example of the `Loop … End` structure is shown in Listing 6-5.

Listing 6-5 Using the Loop … End structure to display a series of numbers

```
Set n 1
Loop
    If "{n}" > 3
        Echo "Done!"
        Break
    End
    Echo "{n}"
    Evaluate n += 1
End
```

The commands in Listing 6-5 define a variable named `{n}` and set its initial value to 1. If the value of `{n}` becomes greater than 3, the loop terminates; otherwise, the current value of `{n}` is displayed in the active window and `{n}` is incremented by 1 for the next loop. Note that the `Break` command terminates the `Loop … End` structure, not the `If` command.

The Begin … End Structure

The `Begin … End` structure groups commands, just as left and right parentheses do. The syntax of the `Begin … End` structure is as follows:

```
Begin
    command …
End
```

Once the MPW Shell executes a `Begin` command, it reads all of the commands between `Begin` and `End` before it executes them. The commands between `Begin` and `End` are processed as a group.

After the `Begin … End` structure, you can enter commands that

- redirect input to the group
- redirect output from the group
- execute only if the commands in the group were successful
- execute only if the commands in the group were not successful

You will find an example of using the `Begin … End` structure in Listing 6-6.

Listing 6-6 Using the Begin … End structure to group commands

```
Begin
    Set
    Export
End > VariableList
```

The example in Listing 6-6 lists all of the MPW Shell variables with their values, writes an export command for each, and then saves the output in a file named VariableList.

The Break and Continue Commands

The `Break` command terminates the nearest `For … End` or `Loop … End` structure. You can add an `If` clause with an expression to a `Break` command to test a condition before terminating the loop. If the expression in the `If` clause is `TRUE`, the enclosing loop is terminated. If you use `Break` alone, without an `If` clause, the loop is terminated unconditionally. The syntax of the `Break` command is

```
Break [If expression]
```

The example in Listing 6-7 shows how to use the `Break` command with an expression to end a structure.

Listing 6-7 Adding the Break command to a structure

```
Set Exit 0
For File in ≈.c
    Print -ps 'Sample PS File' "{File}"
    Break If "{Status}" != 0
    Echo "{File}" is printed.
End
```

The example in Listing 6-7 is similar to the example shown earlier in Listing 6-4, but it provides a way to end the loop if any command in the loop produces an error message, that is, if any command returns a value of `{Status}` other than 0. The value of the `{Status}` variable is changed by each command that is executed. To determine whether any individual command fails, you must either test or save `{Status}` after each command.

The `Continue` command terminates the current iteration of a `For … End` or `Loop … End` structure and allows the structure to continue with its next iteration. For example, Listing 6-8 shows a set of control structures that use the `Continue` command.

Listing 6-8 Adding the Continue command to a structure

```
For i in 1 2 3 4 5
    Echo "{i}"
        If `Evaluate "{i}" == 3`
            Continue
            Echo This is the inner loop
        End
    Echo This is the outer loop
End
```

The lines in Listing 6-8 have an outer `For … End` structure and an inner `If … End` structure. The `For … End` structure has five iterations, because the value of the variable `{i}` can be 1, 2, 3, 4, or 5.

First, the `For … End` structure displays the current value of `{i}`. Then, the `If … End` structure evaluates the current value of `{i}`. If `{i}` is not equal to 3, the MPW Shell skips the remaining commands in the `If … End` structure and then displays

```
This is the outer loop
```

in the active window. If `{i}` is equal to 3, the `Continue` command terminates that iteration of the `For … End` structure. Then, the MPW Shell returns to the `For` command and continues with the next iteration of the `For … End` structure.

The command

```
Echo This is the inner loop
```

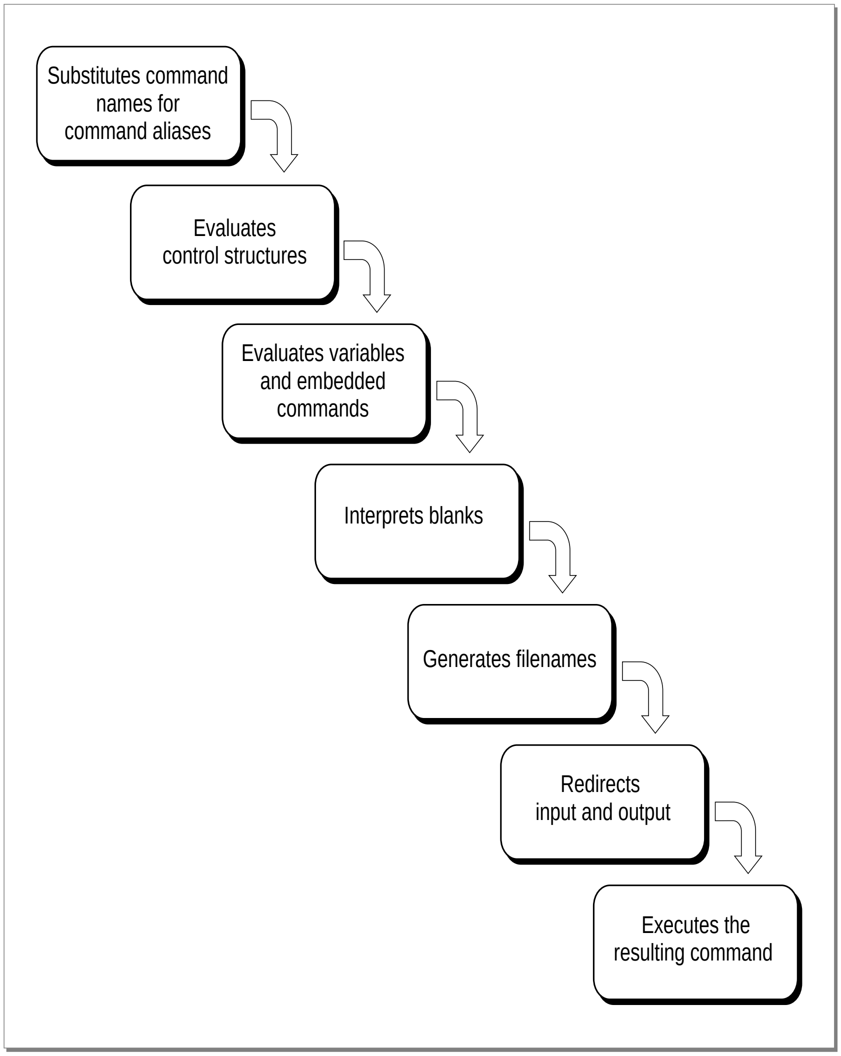
is never executed. If you execute the commands shown in Listing 6-8, the MPW Shell displays the following lines in the active window:

```
1
This is the outer loop
2
This is the outer loop
3
4
This is the outer loop
5
This is the outer loop
```

How Commands Are Interpreted

To interpret a command, the MPW Shell's command interpreter uses a set of steps in a specified order, as illustrated in Figure 6-1.

Figure 6-1 The process of interpreting commands


Substitutes command names for command aliases
Evaluates control structures
Evaluates variables and embedded commands
Interprets blanks
Generates filenames
Redirects input and output
Executes the resulting command

Each step in the process is explained below. When it receives a command, the command interpreter takes the following steps:

1. Substitutes command names for command aliases.

   The first thing the command interpreter does is scan the command for any words that are command aliases. When a command alias is found, it is translated to its original MPW Shell command.

2. Evaluates control structures.

   The MPW Shell has various control structures, including the `Begin … End` structure, the `If … Else … End` structure, the `For … End` structure, and the `Loop … End` structure. If the MPW Shell finds a control structure, it evaluates it at this point.

3. Evaluates variables and substituted commands.

   The command interpreter replaces all variables, whether quoted or not, with their value. It also replaces all commands used as parameters to other commands with their output. If a command contains an ellipsis, the command interpreter opens the appropriate Commando dialog box and uses the resulting command as output.

4. Breaks the command line into a series of “words,” which are separated by spaces or tabs.The following special characters are considered separate words, whether or not they are separated by blanks:

Character	**Meaning**
`;`	Join commands on one line
`|`	Pipe output
`||`	If not successful, do …
`&&`	If successful, do …
`( )`	Group commands or expressions
`<`	Redirect standard input
`>`	Redirect standard output and replace
`>>`	Redirect standard output and append
`≥`	Redirect error messages and replace
`≥≥`	Redirect error messages and append

5. Generates filenames.

If the command interpreter finds a filename pattern, it uses the pattern to generate a list of files. A filename pattern is a word that contains any of the following special characters not marked with quotation marks:

Character	Meaning
?	Match any character
≈	Match any string
[ ]	Specify a scan set
*	Zero or more occurrences
+	One or more occurrences
« »	Specify a repetition

The command interpreter replaces the pattern with a list of filenames that match it, in alphabetical order. If the pattern has no matches, the command interpreter displays an error message.

6. Redirects input and output.

   If the command interpreter finds an input or output redirection character, it interprets the character and prepares to redirect the command's input or output.

Character	Meaning
<	Redirect standard input
>	Redirect standard output and replace
>>	Redirect standard output and append
≥	Redirect error messages and replace
≥≥	Redirect error messages and append
∑	Redirect all output and replace
∑∑	Redirect all output and append

7. Executes the resulting command.

   Any remaining single or double quotation marks are removed, and the command is executed.

A Sample Command

To see how a command is interpreted, it is helpful to break it into parts and examine it. For example, if you first define a command alias to make the command `ls` substitute for the command `Files`, as in

```
Alias ls Files
```

you can then write a command using `ls`, as in

```
ls {MPW}Examples:CExamples:≈.c > List
```

which lists all the files in the CExamples directory whose names end in `.c` and stores them in a file named List. To interpret the command, the command interpreter would take the following steps:

1. Replace command aliases with actual command names. At this step, the command interpreter would replace `ls` with `Files` so that the command becomes

   ```
   Files {MPW}Examples:CExamples:≈.c > List
   ```

2. Evaluate control structures. The sample command has no control structures, so the command interpreter skips this step.
3. Evaluate variables and embedded commands. The sample command has one variable, `{MPW}`, and no embedded commands. The command interpreter expands the variable `{MPW}` so that the command becomes

   ```
   Files Graphics:MPW:Examples:CExamples:≈.c > List
   ```

4. Interpret blank spaces. The command interpreter divides the command into parts by determining where blank spaces are placed. The parts of the sample command are

   ```
   Files
   Graphics:MPW:Examples:CExamples:≈.c
   >
   List
   ```

5. Generate filenames. The command interpreter reads the ≈ character (Option-X) and generates a list of filenames that matches the pattern, similar to this list:

   ```
   Graphics:MPW:Examples:CExamples:Count.c
   Graphics:MPW:Examples:CExamples:Sample.c
   Graphics:MPW:Examples:CExamples:SillyBalls.c
   Graphics:MPW:Examples:CExamples:TESample.c
   ```

6. Redirect input and output. The command interpreter reads the > character, interprets it as redirecting output, and creates a file named List if one does not already exist.
7. Execute the resulting command. At this step, the command interpreter executes each of the parts of the command. As the `Files` command sends its output to the active window, the command interpreter intercepts it and stores it in the file named List.

Using the Parameters Command to Check Parameters

The `Parameters` command helps you understand how a parameter is passed to a command. If you enter `Parameters` with a parameter, the MPW Shell displays the result of the parameter, showing you how it is given to the command interpreter. `Parameters` is useful for debugging commands, especially when you need to understand how parameters within quotation marks are interpreted.

For example, if you enter

```
Parameters ≈
```

the command interpreter evaluates the ≈ character, which matches all the files in the current directory, and lists them in the active window:

```
{0} Parameters
{1} MPW Shell
{2} Examples
{3} Interfaces
{4} Libraries
{5} Tools
{6} Worksheet
```

Note that the first item in the list is the `Parameters` command itself. If you place quotation marks around the ≈ character, as in

```
Parameters '≈'
```

`Parameters` responds with

```
{0} Parameters
{1} ≈
```

to show you that the ≈ character is taken literally. More examples of using the `Parameters` command are listed in Table 6-3.

Table 6-3 How the Parameters command works

Command	Output

`Parameters "{Commands}"`	`{0} parameters`
	`{1} :, Graphics:MPW:Tools:,`
	`      Graphics:MPW:Scripts:`

`Parameters '{Commands}'`	`{0} parameters`
	`{1} {Commands}`

``Parameters `Date` ``	`{0} parameters`
	`{1} Saturday,`
	`{2} October`
	`{3} 17,`
	`{4} 1992`
	`{5} 3:45:39`
	`{6} PM`

``Parameters "`Date`"``	`{0} parameters`
	`{1} Saturday, October 17, 1992 3:47:50 PM`

You can use `Parameters` to test commands before you enter them so that you avoid possible problems. For example, a command such as

```
Delete ≈ .test
```

deletes *all* the files in the current directory and then attempts to delete a file named `.test`. Before you enter such a potentially dangerous command, you can enter

```
Parameters ≈ .test
```

to test how the parameters would be interpreted. In this case, the `Parameters` command would list all the files in the current directory and place the parameter `.test` at the end of the list:

```
{0} Parameters
{1} MPW Shell
{2} Examples
{3} Interfaces
{4} Libraries
{5} Tools
{6} Worksheet
{7} .test
```

Realizing that you do not want to delete the contents of the current directory, you could rewrite the `Delete` command as

```
Delete ≈.test
```

to delete only the files that end in `.test`.

Using Variables and Quotation Marks in Scripts

In the chapter “Entering Commands,” you learned the basic rules of using variables and quotation marks in MPW Shell commands. The rules you learned are probably sufficient for entering commands interactively in the Worksheet window. However, in order to write scripts, you need to learn the more advanced methods of using variables and quotation marks that are described in this section.

Extending the Scope of a Variable

The MPW Shell recognizes variables that are created with the `Set` command only while they are in the current context. For example, if you define a variable by entering a command in the active window, the variable is recognized only in other commands you enter until you leave MPW. Likewise, if you define a variable in a script, the MPW Shell recognizes the variable only in that script.

Sometimes, however, your scripts will call other scripts by using a script name as a command. A script that calls another script is called an **enclosing script** (or sometimes, an outer script). The script that is called is a **nested script** (or sometimes, an inner script). If you define a variable in an enclosing script, you can make the variable known to any nested scripts; to do so, you **export** the variable using the `Export` command. However, if you define a variable in a nested script, the variable is known only within that script and cannot be exported to its enclosing scripts.

The syntax of the `Export` command is

```
Export [name1 [name2]… | -r | -s]
```

For example, you can enter the commands

```
Set Value 2
Export Value
```

to define a variable named `{Value}` and export it. Note that you do not need to place the variable name within `{ }` characters in the `Export` 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 `Unexport` to remove any desired variables from the list. `Unexport` has the same syntax as `Export`.

If you execute an enclosing script with the `Execute` command, as in

```
Execute MarkIt
```

all of the variables that are defined in scripts that are nested within MarkIt are made known to MarkIt.

If you want to define a variable globally so that it is recognized by all scripts and commands, you can define it in and export it from your UserStartup•*name* file. Because the UserStartup•*name* file is executed by the Startup file, all of the variables contained in UserStartup•*name* are defined globally for the MPW Shell.

For example, you could include these commands in your UserStartup•*name* file to make the Courier font the default font for your MPW environment:

```
Set Font Courier
Export Font
```

Building a List of Items in a Variable

A list is one or more strings stored in a variable. Each string in the list is separated from the next by a space. If a string in the list contains spaces or special characters, it is enclosed in single quotation marks.

To create a list of items, you start with the command

```
Set list ""
```

to create a variable named `{list}` and set its value to the null string. Then, for each item you want in the list, enter a command like this one:

```
Set list "{list} '{item}'"
```

to add an item to the list. The MPW Shell expands the variables `{list}` and `{item}` and adds the new result to `{list}`. For example, you could enter

```
Set numberList ""
Set numberList "{numberList} '{1}'"
Set numberList "{numberList} '{2}'"
```

to create a list of numbers containing the numbers 1 and 2. The single quotation marks around each occurrence of `{item}` are treated as literal characters because they are enclosed in double quotation marks. Therefore, each item that is added to the list is enclosed in single quotation marks.

In your script, you can now loop through the items in the list to use them as parameters to another command. Listing 6-9 on page 7-22, taken from the script ReplaceAll, shows an example of this technique.

Listing 6-9 Looping through each item in a list

```
#   Loop through files in the list.
    For fName in {files}

    #   Make file the active file.
        Open "{fName}" ≥ Dev:Null
        Continue If "{Status}" != 0

    #   Position cursor at the top of the file.
        Find • "{fName}"

    #   Replace all instances of the selection with the new string.
        Replace -c ∞ /{Old}/ "{New}" "{fName}"

    #   Close the file; save changes.
        Close -y "{fName}"
    End
```

In Listing 6-9, the variable `{Files}` contains the names of the files in which the script performs its find-and-replace operation. The script opens the first file, making its window the active window and sending any error messages to the virtual device `Dev:Null`.

Next, the script places the insertion point at the top of the file, finds the old selection, replaces it with the new string, and then closes the file, saving any changes that were made. The script then continues with the next file in the list. Notice that the `For … End` loop contains a `Continue If` statement that ends the current iteration of the loop if an error occurs when the script attempts to open any file.

Using Quotation Marks With List Variables

In an MPW Shell command, you should usually place variable names within double quotation marks. However, when a variable contains a list of items, you omit the quotation marks so that the list is interpreted as a number of individual items separated by spaces. If you use double quotation marks around a variable that contains a list, the list would be treated as one long string.

For example, the following command (taken from the script MyPrint) contains two variables, one with quotation marks and one without:

```
Print {List} -font Monaco -size 10 -h -ps "{PSFile}"
```

Note that the variable `{List}` is not placed in quotation marks because it contains a list of the files to be printed, defined earlier in the script.

However, quotation marks are required around the variable `{PSFile}` because it contains the name of a PostScript file that `Print` will use to improve the appearance of the printed output, and the filename might contain spaces or special characters.

Listing 6-9 is another example that shows how you can use variables with and without quotation marks. In the listing, the variable `{files}` has been assigned a list of files in which to search and replace a text string. Because each file in the list should be treated as an individual item, `{files}` is not enclosed in quotation marks.

As the files are processed, the variable `{fName}` contains the name of the current file. Therefore, `{fName}` is enclosed in quotation marks because the filename may contain spaces or special characters and should be treated as one long string.

Nesting Embedded Commands

To nest an embedded command within another embedded command, mark each of the inner backquotes with a ∂ character. The ∂ character preceding each inner backquote indicates that the left inner backquote is not to be paired with the left outer backquote.

For example, the following command is taken from the scripts CapFirst, ProtoGen, and MarkIt:

```
Set wasOpen `Evaluate "∂`Windows∂`" =~ /≈{fullName}≈/`
```

Note that the embedded command

```
"∂`Windows∂`"
```

is nested within the embedded `Evaluate` command.

In the command, the output of `Windows`, which is a list of the windows currently open, is passed as a parameter to `Evaluate`. `Evaluate` then compares the list of windows to the value of `{fullName}`. (The variable `{fullName}` has already been defined as the full pathname of a file.)

If the pathname defined in `{fullName}` is found in the list of windows, the `Set` command sets `{wasOpen}` to 1, or `TRUE`. Otherwise, it sets `{wasOpen}` to 0, or `FALSE`. The value of `{wasOpen}` is checked later in the script to determine whether or not to close the window.

Using Quotation Marks With Embedded Commands

Using quotation marks with embedded commands is similar to using them with variables. In general, if the embedded command returns a list, you should not place it in quotation marks. If it returns single items, you may or may not need quotation marks, depending on whether the command itself places the values in quotation marks.

If an embedded command returns a list of items that should be treated individually, you should not place the command within quotation marks. For example, the `Windows` command displays a list of all of the open MPW Shell windows. You can place `Windows` inside backquotes and use it as an embedded command, as in Listing 6-10, taken from the script CleanWindows.

Listing 6-10 Using an embedded command to return a list of items

```
#   Loop through list of open windows.
    For F in `Windows`
        Movewindow 2 2 "{F}"
        Sizewindow 512 386 "{F}"
    End
```

In Listing 6-10, the embedded command `` `Windows` `` is not placed inside quotation marks, because you want each item in the list to be treated individually. Furthermore, the `Windows` command itself places single quotation marks around those window names that contain spaces or special characters.

However, you can also use an embedded command to set the value of a variable to a list of items. If you do, you should place the embedded command inside double quotation marks so that the enclosing `Set` command treats the list as one long string. For example, the commands

```
Set MyWindows "`Windows`"
For F in {MyWindows}
    MoveWindow 2 2 "{F}"
    SizeWindow 512 386 "{F}"
End
```

set the variable `{MyWindows}` to the list of open windows created by the `Windows` command. In this case, `` `Windows` `` is placed inside double quotation marks, because it returns a list of window names.

The MPW Shell commands differ in how they return text strings that contain spaces. Some commands, such as `Windows`, mark them with single quotation marks; other commands do not mark them at all.

If you use an embedded command that returns a single item, check whether the command itself places quotation marks around text strings with spaces or special characters. To check the behavior of a command, refer to the *MPW Command Reference*.

If the embedded command returns values without single quotation marks, add your own double quotation marks outside the backquotes. For example, in the command

```
Set File "`Request 'Open what file?'`"
```

the `Request` command is placed in double quotation marks. `Request` displays a dialog box asking the user to enter a filename. Because the user might enter a filename that contains spaces that you want treated as a single string, the double quotation marks around `Request` are required. (Note that the message `Open what file?` is placed inside quotation marks because it contains two spaces and a special character.)

However, if the embedded command returns values with single quotation marks, you should not add double quotation marks to it. If you do, the single quotation marks the command returns are treated as literal characters.

Preventing Quotation Marks From Being Discarded

When you place a variable or an embedded command inside double quotation marks, the command interpreter removes the quotation marks when the variable or embedded command is expanded. For example, the command

```
Print "{PFile}"
```

expands the variable `{PFile}`, discards the quotation marks, and prints the file whose name is contained in `{PFile}`.

When you define a variable to have a value that contains quotation marks (also called inner quotation marks), you must

- use additional, outer quotation marks when you define the variable
- place the variable within `{{` and `}}` characters when you obtain its value, so that the inner quotation marks are not discarded

For example, if you enter

```
Set Alpha "we'll"
```

you define a variable named `{Alpha}` and set its value to the string

```
we'll
```

Note that you must place double quotation marks around the value when you define it, because the value of the variable contains an inner, single quotation mark. The double quotation marks are not part of the value itself.

Once the variable `{Alpha}` is defined, if you enter the command

```
Echo {Alpha}
```

to obtain the value of `{Alpha}`, the MPW Shell displays only the error message

```
*** MPW Shell - 's must occur in pairs.
```

But if you enter the same command with the `{{` and `}}` characters instead of `{` and `}` characters, as in

```
Echo {{Alpha}}
```

the MPW Shell displays the value of the variable, like this:

```
we'll
```

The same rules apply when the value of a variable contains a pair of quotation marks. For example, if you enter

```
Set Beta '"hello"'
Echo {{Beta}}
```

the MPW Shell discards the outer quotation marks and responds with the value of `{Beta}`, which is

```
"hello"
```

To prevent quotation marks from being discarded from the output of an embedded command, place a `` character (instead of a ` character) on either side of the embedded command.

For example, suppose you have a file named Example that contains only one line:

```
This is a single quote '
```

First, you would use the `Set` command to define a variable that represents the contents of the file. The first part of the command would look like this:

```
Set myFile
```

Now the command needs a definition for the variable. To define the variable, you use the `Catenate` command, which writes the contents of the file, and you place a `` character on either side of the command, like this:

```
Set myFile ``Catenate Example``
```

Because the `Catenate` command is within `` characters (also known as double backquotes), the quotation mark the file contains is not discarded. Note that the entire command is placed within double quotation marks, because it functions as a two-word parameter to the `Set` command.

If you then display the contents of the variable with the `Echo` command, using `{{` and `}}` characters around the variable, as in

```
Echo {{myFile}}
```

the MPW Shell responds with

```
This is a single quote '
```

saving the quotation marks the file contains.

Combining Commands Into Scripts

You can write scripts that perform many different types of tasks. In general, however, scripts usually perform these basic functions:

- handle any parameters that the user provides
- initialize variables
- do the main part of its work
- check for and handle errors as it works
- perform limited cleanup and close the script

You should begin your script with commands that establish what will happen if an error occurs while the script is running. In most cases, you would specify that if an error occurs, your script handles the error and continues running. In addition, to make debugging your scripts easier, you can enter a command that displays a message containing the location of the error. Writing the commands that begin a script is described in the section "Beginning Your Script" on page 7-28.

Next, you can define and initialize the variables the script will use to do its work. In addition, you may want your user to specify parameters when he or she starts the script. The parameters your user specifies can be file or directory names, regular expressions, options that begin with a hyphen, and so on. When a user provides these parameters, your script needs to handle them appropriately. Handling parameters is described in the section "Handling Parameters in Scripts" on page 7-28.

Now you can write the commands that do the main work of the script. Your script can open files, manipulate windows on the screen, search for and edit text strings, find a line number in a file, display alert and dialog boxes for users, and add menus to the MPW Shell menu bar. You can refer to the sample scripts in the MPW Script Tips folder to see how to write commands and control structures that perform different tasks. To see the different types of alert and dialog boxes your script can display, and to learn how to display them, you can read the section "Using Alert and Dialog Boxes in Scripts" on page 7-32.

As you write the body of your script, you should add commands that check for errors and handle them. One technique you can use frequently is to check the value of the variable `{Status}`. The `{Status}` variable contains a number indicating whether the last command that was executed was successful (a value of 0) or caused an error (any other value). You should add commands to your scripts that handle errors that might occur, as explained in the section "Handling Errors in Scripts" on page 7-39.

Once your script has done its main work, it should perform its cleanup work and close. The details that you may want to handle at this part of the script include closing any files and windows that the script has opened, prompting the user to save the changes that have been made to a file, and setting the `{Status}` variable so that it contains a code that shows whether the script executed successfully.

As with any other source file you write, you should remember to document your scripts so that other developers can refer to or update them. You may also want to create Commando dialog boxes for your scripts, so that others can access them by using the familiar Macintosh interface. Creating Commando dialog boxes is described in detail in *Building and Managing Programs in MPW*.

Beginning Your Script

You can begin your scripts with these commands:

```
Set Exit 0
Export Exit
Set TraceFailures 1
Export TraceFailures
```

The first two commands set the variable `{Exit}` to 0 so that the script does not terminate upon error, and then export that value of `{Exit}` to any nested scripts. Once you set `{Exit}` to 0, your script terminates only if you add an explicit `{Exit}` command, or if it has no more commands to execute. If your script calls other scripts, and if any of the scripts produces an error, all of the scripts continue running.

The second `Set` command sets the variable `{TraceFailures}` to 1 so that if an error occurs, the MPW Shell displays a message that locates the error. The `Export` command extends the scope of the `TraceFailures` variable so that it is recognized in any scripts that are nested within the script you are writing.

Using the information that {TraceFailures} provides is described in the section "Debugging Your Scripts" on page 7-43.

Handling Parameters in Scripts

To handle parameters to a script name, the MPW Shell provides special variables known as parameter variables. **Parameter variables** allow your script to access the parameters the user entered to start the script. For example, to use the script Hexify, a user might enter

```
Hexify sample1 -t textFile -d
```

to specify the filename and style of printing. Your script must be able to handle the parameters your user entered.

You use the parameter variables in a group. The first variable in the group is {#}, which contains the number of parameters the user entered. In the example just given, the value of {#} would be 4. Thereafter, the variables are numbered {0}, {1}, {2}, and so on. The value of the variable {0} is the script name. The value of {1} is the first parameter the user entered; the value of {2} is the second parameter the user entered, and so on. The variable {Parameters} contains a list of all the parameters the user entered, as shown in Table 6-4.

Table 6-4 The parameter variables

Variable	Value
{#}	The number of parameters the user entered
{0}	The script name
{1}, {2}, …, {*n*}	The first, second, and *n*th parameter the user entered
{Parameters}	The value of {1}, {2}, and {*n*}
{"Parameters"}	The value of {1}, {2}, and {*n*}, with each value enclosed in quotation marks

So, if a user started Hexify with the command

```
Hexify sample1 -t textFile -d
```

the parameter variables passed to Hexify would have the values shown in Table 6-5.

Table 6-5 The values of the parameter variables

Variable	Value
`{#}`	`4`
`{0}`	`Hexify`
`{1}`	`sample1`
`{2}`	`-t`
`{3}`	`textFile`
`{4}`	`-d`
`{Parameters}`	`sample1  -t  textFile  -d`
`{"Parameters"}`	`"sample1"  "-t"  "textFile"  "-d"`

In your scripts, you can choose to interpret the parameter variables in one of the following ways:

- You can require just one parameter.
- You can require that your user enter the parameters in a fixed order on the command line, and then refer to them by number.
- You can preface a parameter with an option, as in

  ```
  Make -f makefile
  ```
- You can use the parameter variables in a command-line parser, so that your user can enter them in any order.

When Parameters Are Specified in a Fixed Order

The script RenameProjFile, which renames a file checked in to Projector, is an example of a script that requires the user to enter parameters in a certain order. For example, to start RenameProjFile, the user must enter a command something like this:

```
RenameProjFile SillyBalls.c PaintBalls.c
```

In the command, the old filename must come first, and the new filename must follow it.

In this case, the script can handle the parameters that were entered by setting variables for the old name and the new name to the parameter variables `{1}` and `{2}`, as shown in Listing 6-11, which is taken from the script RenameProjFile.

Listing 6-11 Setting variables to handle parameters entered in a fixed order

```
#   Set variables to specified files.
    Set oldName "{1}"
    Set newName "{2}"
```

Note that this method of handling parameters does not allow for much flexibility. If the filenames were entered in reverse order, the script would not work correctly.

When Parameters Consist of Just One Word

The example shown in Listing 6-12, taken from the script GoTo, demonstrates how your script can handle parameters of one word.

The commands in Listing 6-12 check each item in `{Parameters}` to determine whether it is a line number, consisting solely of numerals, or a window name. If the parameter is a line number, the variable `{LineNum}` is set; if the parameter is a window name, the variable `{Window}` is set.

Listing 6-12 Processing simple parameters

```
#   Loop through parameters.
    For Item in {"Parameters"}
        #   If the parameter consists of numbers, it is
        #   the line number.
            If "{Item}" =~ /[0-9]+/
                If {LineNum} == ""
                    Set LineNum "{Item}"
                Else
                    Echo "### Line number multiply defined."
                    Echo "### Usage: {0} [LineNumber] [Window]"
                    Exit 1
                End
        #   Otherwise, the parameter is the name of the window.
            Else
                If {Window} == ""
                    Set Window "{Item}"
                Else
                    Echo "### Window multiply defined."
                    Echo "### Usage: {0} [LineNumber] [Window]"
                    Exit 1
                End

            End >> Dev:StdErr
    End
```

When Parameters Consist of More Than One Word

When the parameters to a script name consist of more than one word, for example, options that themselves take a parameter such as `-o` *filename*, use the `Loop … End` structure, which can process more complex parameters. The `Loop … End` structure is an infinite loop, unless a `Break` command terminates it. (`Break` terminates the loop if a certain condition is met.)

The example shown in Listing 6-13 is taken from the script ReplaceAll, which replaces all occurrences of a text pattern with a text string in the files whose names are specified.

Listing 6-13 Processing complex parameters

```
#   Save pattern to find and replace in Old.
    Set Old "{1}"

#   Save string to replace it with in New.
    Set New "{2}"

#   Shift parameters by 2 so that parameters are now all
#   file names or directories.
    Shift 2
```

```
#   Loop through remaining parameters.
    For Item in {"parameters"}
    # [commands follow that work on each item in the
    # parameter list ...]
    End
```

To start the script, you must enter parameters to the ReplaceAll command in a certain order. First, you enter a selection expression that specifies a pattern to be replaced. Then, you enter a text string that is used as the replacement. Last, you enter a list of files or directories in which the string will be replaced.

Thus, you can start ReplaceAll with a command like this one:

```
ReplaceAll /first/ "second" sample1 sample2 sample3
```

If you use this command, the parameter variables contain the following values:

```
{#}             5
{0}             ReplaceAll
{1}             /first/
{2}             "second"
{3}             sample1
{4}             sample2
{5}             sample3
{Parameters}    /first/  "second"  sample1  sample2  sample3
```

The commands in Listing 6-13 save the pattern to be replaced in the variable `{Old}` and the string that replaces it in the variable `{New}`. Then, they use the `Shift` command to renumber the parameters, decreasing the value of each by 2 so that `{3}` becomes `{1}`, `{4}` becomes `{2}`, and `{5}` becomes `{3}`. This way, the script can work on all of the parameters numbered 3 and higher, that is, on all of the file and directory names.

You will have a chance to practice writing the ReplaceAll script in the section "Tutorial—Building a Script" on page 7-44.

Using Alert and Dialog Boxes in Scripts

You can use MPW Shell commands to display various types of alert and dialog boxes. In a script, alert and dialog boxes can

- request information from users
- allow users to decide whether the script should execute a certain command
- alert your users to errors

The types of alert and dialog boxes you can use are described in the following sections.

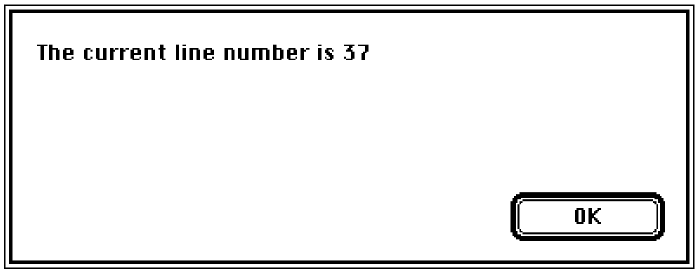
Adding an Alert Box

An alert box displays a message to your user and can have one or more buttons. Alert boxes are useful when you want to make a user aware of an error or specific output, but do not require the user to do anything other than read the message and click a button. Alert boxes that you create with the `Alert` command look like the one shown in Figure 6-2.

The `Alert` command has the general syntax

```
Alert [message] [-s]
```

Figure 6-2 An alert box created with Alert


The current line number is 37

With the `Alert` command, you can either enter a text string as a message, or you can specify the name of a file that contains the message and that will be used as input.

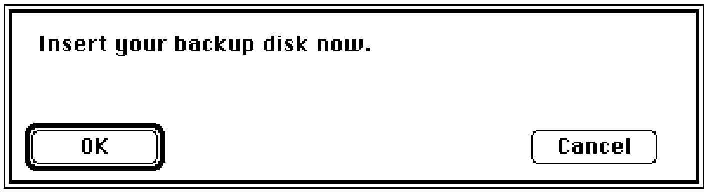
To display the alert box shown in Figure 6-2, you would enter

```
Alert "The current line number is `Position -l {Window}`"
```

The `Position` command finds the current line number, and the result is passed to the `Alert` command. By default, the `Alert` command causes a beep when the alert box is displayed. To disable the beep, add the `-s` option to the command.

To display an alert box with two buttons, an OK button and a Cancel button, as shown in Figure 6-3, you use the `Confirm` command.

Figure 6-3 An alert box created with Confirm


Insert your backup disk now.

The general syntax of `Confirm` is

```
Confirm [message] [-t]
```

Just as with the `Alert` command, you can either enter a specific message with `Confirm`, or you can specify the name of a file that contains the message and that will be used as input. To display the two-button alert box shown in Figure 6-3, you use `Confirm` without its `-t` option, as in the command

```
Confirm 'Insert your backup disk now.'
```

When the alert box appears, the user can insert a backup disk and click OK or click Cancel. The lines shown in Listing 6-14 show how the `Confirm` command without its `-t` option is used in the script Quit•Mine.

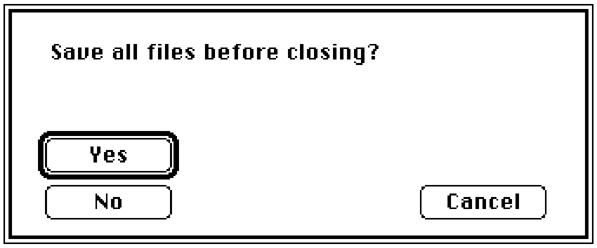
Listing 6-14 Using Confirm in a script

```
#   Prevent script from aborting if a command returns a
#   nonzero status code.
    Set Exit 0

#   Prompt user to insert backup disk.
```

```
    Confirm "Insert backup disk."

    #   OK is selected.
        If "{Status}" == 0
        #   Execute the backup commands here ...

    #   Cancel is selected.
        Else
            Echo "# {0}: Backup cancelled."
        End
```

To display an alert box with three buttons (a Yes button, a No button, and a Cancel button), use `Confirm` with its `-t` option. A three-button alert box is shown in Figure 6-4.

Figure 6-4 An alert box created with Confirm -t


Save all files before closing?

To display the alert box shown in Figure 6-4, you would enter

```
Confirm -t 'Save all files before closing?'
```

An example showing how you might use `Confirm -t` in a script is presented in Listing 6-15.

Listing 6-15 Using Confirm -t in a script

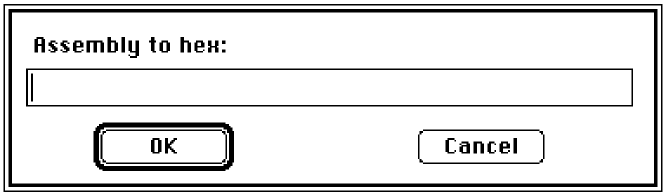
```
#   Prevent the script from aborting if a command
#   returns a non-zero status code
    Set Exit 0
```

```
#   Display confirm dialog
    Confirm -t "Save all files before beginning?"
#   Save status.
    Set myStatus "{Status}"
#   Exit script if the Cancel button was chosen.
    Exit If "{myStatus}" == 5
#   Save all files if the Yes button was chosen.
    If "{myStatus}" == 0
        Save -a
    End
```

Adding Dialog Boxes That Contain Text Boxes

To display a dialog box that contains a text box, an OK button, and a Cancel button, use the `Request` command. The dialog box shown in Figure 6-5 asks a user to enter a line of assembly code. Your user can either type something in the text box and click OK or click Cancel.

Figure 6-5 A dialog box that contains a text box



`Request` has the general syntax

Request [*message*] [-d *default*] [-q]

With the `Request` command, you can specify an exact message, or you can specify the name of a file that contains a message. To display the dialog box shown in Figure 6-5, you would enter

```
Request 'Assembly to hex:'
```

without using the `-q` or `-d` options. (The `-q` option specifies that the `{Status}` variable is not changed if the user clicks the Cancel button. The `-d` option specifies a default response that is displayed in the text box when the dialog box first appears. When the user clicks OK, the default text is returned to the command, unless the user has changed the text.)

When your user enters text in the editable field, your script must handle the input. The example in Listing 6-16, taken from the script A2Hex, shows how your script might do so.

Listing 6-16 Handling input from a dialog box

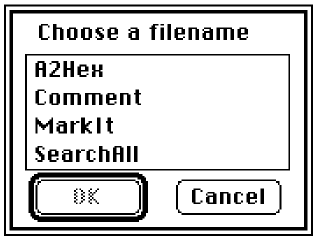
```
#   Request a line of assembly code.
    Set assembly "`Request "Assembly to hex:"`"

#   If Cancel button was chosen, delete temporary file
#   and exit script.

    If "{assembly}" == ""
        Delete {0}Temp.a -i
        Exit 0
    End
```

The example in Listing 6-16 defines a variable named `{Assembly}` and sets its value to the response the user types in the “Assembly to hex:” text box. If the user enters nothing or clicks Cancel, the value of `{Assembly}` is set to the null string. If the value of `{Assembly}` is the null string, the script deletes the temporary file that was created and then terminates.

Adding Dialog Boxes With Lists

To display a dialog box with a list, an OK button, and a Cancel button, use the `GetListItem` command.

Figure 6-6 A dialog box with a list



`GetListItem` has the general syntax

```
GetListItem [item1] [item2]… [-d item] [-m message] [-s] …
```

The items you specify in the command are the items that appear in the list. Your user can select any one of these. The `-d` option, followed by the name of an item in the list, specifies that the item will be selected by default when the list first appears. The `-m` option, followed by a message, prints the message above the list. The `-s` option specifies that a user can select only one item from the list and allows the user to scroll through the list using the cursor keys.

To create the dialog box shown in Figure 6-6, you would enter the command

```
GetListItem -m "Choose a filename" `Files` -s
```

In a script, you can use a more complex command to display a dialog box with a list, like this one:

```
Set Window "`GetListItem ∂`Windows∂` -m 'Choose a window' -s `"
```

To understand this complex command, start from the innermost element and move outward:

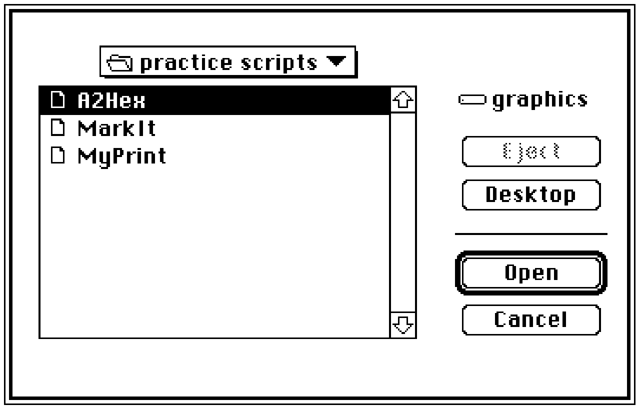
1. The `∂`Windows∂`` command creates a list of the windows that are open and passes the list to `GetListItem`.
2. `GetListItem` receives the output of `Windows` and displays a dialog box with a list of all the open windows. `GetListItem` uses the option `-m` to display a message above the list and the option `-s` to display the list in single-selection mode.

3. The variable `{Window}` is assigned the window name the user selects from the list.
4. The script continues, using the value of `{Window}`.

Adding Standard File Dialog Boxes

To display a standard file dialog box like the one shown in Figure 6-7, you can use the `GetFileName` command.

Figure 6-7 A typical standard file dialog box



`GetFileName` has the general syntax

GetFileName [*pathname*] [-b *buttonTitle*] [-d] [-m *message*] [-s] …

By default, the standard file dialog box that `GetFileName` displays contains buttons labeled Eject, Desktop, Open, and Cancel, and allows your user to select a file from the list. The `-b` option, followed by a button title, renames the default button (in Figure 6-7, the default button is the Open button, because it is highlighted).

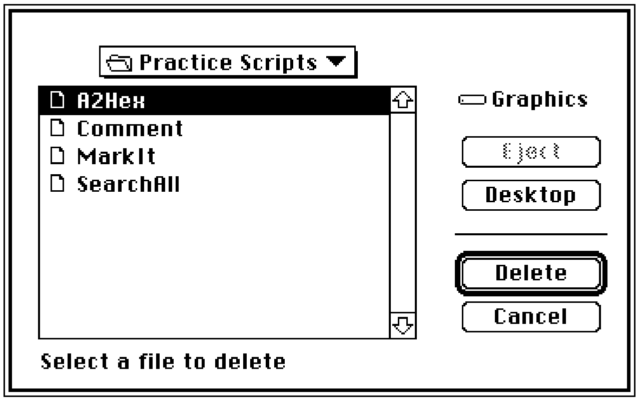
The `-d` option displays a standard file dialog box that allows your user to select a direc- tory and contains a Directory button in addition to the buttons on a typical standard file dialog box. The `-m` option, followed by a message, specifies a message that is displayed below the file list. The `-s` option specifies that `GetFileName` returns a status of 0, even if your user clicks Cancel.

For example, the command

```
Set File "`GetFileName -m 'Select a file to delete' -b Delete -s`"
```

from the script AddDeleteMenu displays the dialog box shown in Figure 6-8. The dialog box displays the prompt “Select a file to delete” and has a button named Delete as its default button.

Figure 6-8 A standard file dialog box with a prompt and a Delete button



The `Set` command is interpreted as follows:

1. `GetFileName` displays a dialog box like the one shown in Figure 6-8.
2. The user chooses a file from the list and clicks Delete.
3. The file becomes the output of `GetFileName` and a parameter to `Set`.
4. `Set` sets the value of the variable `{File}` to the file the user selected.

Once the `Set` command is interpreted, the script continues, using the value of `{File}`.

Handling Errors in Scripts

When an error occurs in a script, it can be a syntax error, a command error, or a usage error.

Syntax errors occur when one of the commands in your script fails because you have not used the command's syntax correctly, for example, if you have left out a required quotation mark. **Command errors** occur when a command does not work as you expect it to. For example, a command in your script might try to open a file that does not exist.

Usage errors, however, occur when the command line your user enters to start the script is incorrect. The user might enter too many parameters, too few parameters, or an illegal parameter. To be reliable, your scripts need to handle all three kinds of errors.

How to Detect an Error in Your Script

You probably have already entered a command at the beginning of your script like this one:

```
Set Exit 0
```

Once the value of `{Exit}` is set to 0, your script needs to be able to detect when a command error occurs. You can usually do this by checking the variable `{Status}`, which contains the status code of the command that was last executed. For most commands, a status code of 0 means that the command was successfully executed. Status codes are determined individually for each command. You can check the meaning of the status codes a command provides in its command page in the *MPW Command Reference*.

Note
When you use the variable `{Status}` in a command, you should place it within double quotation marks. If a command is not successful, it returns a negative value such as –2 to `{Status}`. In a negative value, the – character is a special character and must be quoted. ◆

The example in Listing 6-17, taken from the script MarkIt, attempts to open a file named `{fName}`, and then checks the value of `{Status}`. If the `Open` command is successful, the value of `{Status}` is 0.

Listing 6-17 Detecting an error by checking the {Status} variable

```
#   Open the file fName. If the file can't be opened, print
#   an error message and exit script.
    Open "{fName}" ≥ Dev:Null
    If "{Status}" != 0
        Echo "### {0}: File {fName} not found."
        Continue
    End >> Dev:StdErr
```

The example uses two virtual devices, `Dev:Null` and `Dev:StdErr`. Virtual devices are sources for input or destinations for output that do not have corresponding physical devices. The pathname `Dev:Null` represents an empty output stream, sometimes called the "bit bucket," and is used for discarding unwanted output. The pathname `Dev:StdErr` represents the current standard error output.

To handle an error without explicitly checking the value of `{Status}`, you can use commands like those shown in Listing 6-18, taken from the script RevertToVersion.

Listing 6-18 Detecting an error without checking the {Status} variable

```
#   Check out the specified revision of the specified file.
#   If Checkout is not successful, exit script.
    CheckOut "{file},{revision}" || Exit 2
```

The command line in Listing 6-18 attempts to check out a file from the Projector database, and if it cannot check out the file, terminates the script. The `||` operator in the command line indicates that the `Exit` command should be executed only if the `CheckOut` command is not successful. The `2` after the `Exit` command is the status code that the script returns if the checkout is not successful.

You can also use the `||` operator in more complex commands, as shown in Listing 6-19, taken from the script Comment.

Listing 6-19 A more complex example of detecting errors

```
#   Set nLines to the number of lines of selected text in
#   the input file. All diagnostic output is discarded.
#   The Count command will fail if Window is not a valid
#   window name. The Echo and Execute commands will be executed
#   if and only if the Count command fails. In that case, an
#   error message is written to standard output and the
#   script is exited.
    Set nLines `(count -l "{Window}.§" ≥ Dev:Null) || ∂
        (Echo "### {0}:{Window} not a valid window name." >> ∂
        Dev:StdErr; Exit 1)` ≥ Dev:Null
```

The commands in Listing 6-19

- count the number of lines of selected text in the window specified by the variable `{Window}`
- set the variable `{nLines}` to the total
- discard any diagnostic output
- display an error message and terminate the script, if the `Count` or `Set` commands fail

Notice that the previous command also uses the virtual devices `Dev:Null` and `Dev:StdErr`.

In general, any value of `{Status}` that is not 0 represents an error. However, you may need to check the meaning of a nonzero value. Table 6-6 summarizes the status codes that have negative values, which indicate an error from the MPW Shell; these errors are not specific to individual commands.

Table 6-6 The status codes with negative values

Return code	Meaning
–1	The command was not found, the script is a directory, the script is not executable, or the script has a bad date.
–2	The filename expansion failed, or an error occurred in the syntax of an expression.
–3	An error occurred in the control structures, quotation characters or braces were not balanced, or an `End` command or `)` character is missing.
–4	A filename following a redirection character is missing, or the file could not be opened.
–5	An invalid expression; used especially with commands such as `If`, `Break`, `Continue`, and so on.
–6	The tool could not be started.
–7	A run-time error, most likely an out-of-memory error, occurred while the tool was executing.
–9	The tool or script was aborted when you pressed Command-period, or a serious internal error occurred. Any script that was previously running is also aborted.

What Your Script Should Do If an Error Occurs

One way to handle an error is simply to terminate the script. Once you have set `{Exit}` to `0`, you can terminate the script with the command

```
Exit If "{Status}" != 0
```

In many cases, you want your script to display an error message before terminating. To do so, you use the `Echo` command, redirecting the output to standard error. For example, Listing 6-20, taken from the script WhatLine, checks the number of parameters the user entered on the command line. If the user enters more than 1, the script displays an error message and terminates.

Listing 6-20 Checking the number of parameters given on the command line

```
#   If more than one parameter is given, write an error
#   message and exit the script.
    If "{#}" > 1
        Echo "### Usage: {0} [Window]"
        Exit 1
    End >> Dev:StdErr
```

Another example of checking for syntax errors, taken from the script Hexify, is shown in Listing 6-21.

Listing 6-21 Checking for repeated options on the command line

```
#   If -d option is specified, want to delete StuffIt files.
    If "{1}" == '-d'
        #   Make sure -d has not already been given as an option.
        If "{D}" == ""
            Set D 1
        #   If it has, write error message and exit script.
        Else
            Echo "## "{0}": Option ∂"{1}∂" multiply defined."
            Echo "## Usage: Hexify name [-t textfile] [-d] [-p] [-r]"
            Exit 1
        End
    #   [Examine some more options and execute commands ...]
    End
```

When the user enters an option on the command line, the script checks it to make sure that the option has not already been given. If it has been given, the script displays several error messages, redirects any output to standard error, and terminates.

Sometimes you do not want your script to terminate, even if it encounters an error. For example, if your script is processing a list of files and finds that one of the files does not exist, you probably want the script to process the rest of the files in the list.

The example shown in Listing 6-22, taken from the script SearchAll, starts by checking whether the current file or directory exists. If it does not, the `Else` part of the command handles the error. Since SearchAll accepts a list of files and directories to search, the `Continue` command instructs the script to continue the loop, searching the next file in the list.

Listing 6-22 Handling an error and continuing

```
#   Loop through list of path names.
    For path in {pathName}

    #   If the pathname is valid, write progress info to
    #   standard output, and recursively search all files and
    #   folders within the path.
        If "`Exists "{path}"`"
            Echo "# Searching ∂"{path}∂" for {Pattern}"
            Search "{Pattern}" `files -t TEXT -r -f "{path}"` ∂
                "{Options}" ≥ Dev:Null
            Echo

    #   Otherwise, write an error message and go on to
    #   next iteration of loop.
        Else
            Echo "### {0}: {path} is not a valid pathname.∂n" ∂
                >> Dev:StdErr
        End
    End
```

Debugging Your Scripts

It is important to debug your scripts, so that your users do not encounter errors. If you have entered the commands

```
Set Exit 0
Export Exit
Set TraceFailures 1
Export TraceFailures
```

at the beginning of your script, you can easily locate the line and character position of the error.

If your script exits with a nonzero error code, the MPW Shell displays the script name and the character position where the error occurred, like this:

```
#---------------------------------------------------------
        File "Graphics:MPW:MarkIt"; Line •!1405
#---------------------------------------------------------
```

If you place the insertion point within the message and press Enter, the script opens, with the insertion point placed just after the character at which the error occurred.

The first part of the message lists the pathname of the script that contains the error. In the second part of the message, `Line` is a command alias for the `Find` command. The regular expression that follows `Line` shows the character position of the error.

Once you locate the error, you can debug it. To fix the bug, you may need to refer to

- the rules that govern how commands are interpreted, described in the section "How Commands Are Interpreted" earlier in this chapter
- the rules of selection expressions and regular expressions, described in the chapter "Editing With Commands"
- the rules of using quotation marks in MPW Shell commands, described in this chapter and in the chapter "Entering Commands"

Another technique you can use to debug your scripts is to add the commands

```
Set Echo 1
Export Echo
```

to the beginning of your script. Then, when you execute the script, its commands will be displayed one by one in the active window. (Or, if you prefer, you can write the commands to redirect the output to a file.) When an error occurs, the script terminates. The line that generated the error is the one that immediately precedes the error message. You should remove or comment these lines before giving the script to others to use.

Tutorial—Building a Script

This tutorial explains how to build the sample script ReplaceAll. The entire script is given here, step by step.

ReplaceAll locates all occurrences of a selection, in all of the files your user specifies, and replaces them with a new string. Your user can specify a directory name, which causes ReplaceAll to work in all of the files that directory and its subdirectories contain.

To complete this tutorial, read each step, try to write the commands it asks for, and then check your answer against the sample commands that are given. If you have questions, you can read the explanation of each step, or you can refer to the appropriate section in this chapter. As you complete each step, be sure to document what you have done.

Note
This is probably the most challenging tutorial in this guide. Don't be discouraged if you do not get each step exactly right. If, however, you complete this tutorial with flying colors, you may be eligible to become an MPW Shell scripting expert. ◆

To build the script ReplaceAll, follow these steps:

1. **Open a new file and name it ReplaceAll. Move ReplaceAll to the MPW directory.**

   This will be the file in which you will enter the commands that make up the script.

2. **Begin the script by specifying that the script should not terminate if an error occurs.**

```
#   Don't exit program on error.
    Set Exit 0
    Export Exit
```

   When the variable `{Exit}` is set to 0, the script continues running even if a command within the script returns a status code other than 0. If you do not set `{Exit}` to 0, the script would terminate. These lines also export `{Exit}`, so that if the script has nested scripts, and an error occurs in any script, all scripts continue running.

   Even though you have set `{Exit}` to 0, you can enter an explicit `Exit` command later in your script that terminates if a certain type of error occurs.

3. **Set the value of the variable `{TraceFailures}` so that you can locate an error if one occurs.**

```
#   Provide error detection.
    Set TraceFailures 1
    Export TraceFailures
```

   If `{TraceFailures}` is set to 1 and an error occurs, the MPW Shell displays a message giving the character position of the error so that you can locate it easily. Because you export `{TraceFailures}`, it is recognized in any scripts that might be nested within ReplaceAll.

   Even if ReplaceAll does not have any nested scripts, the `Export` command does not do any harm.

4. **Write a control construct to check the parameters your user has entered. There should be at least three—a selection, a replacement string, and a file or directory name.**

```
#   Make sure there are at least 3 parameters.
    If "{#}" < 3
        Echo "### {0}: too few parameters"
        Echo "### Usage: {0} selection newstring files..."
        Exit 1
    End
```

   The control structure checks the value of the `{#}` variable, which is a parameter variable that shows the number of parameters the user entered. If the user has entered only one or two parameters, the control structure displays an error message and syntax message, and then terminates, passing a status code of 1 to the `{Status}` variable.

5. **Define a variable named Old that will store the pattern to be replaced.**

```
#   Save pattern to find and replace in Old.
    Set Old "{1}"
```

   This `Set` command sets the value of `{Old}` to the parameter variable `{1}`, which is the first parameter the user entered to start the script. Notice that you must place `{1}` within double quotation marks, because its value is a selection expression that could contain spaces or special characters.

6. **Define a variable named New that will store the replacement string.**

```
#   Save string to replace it with in New.
    Set New "{2}"
```

This `Set` command sets the value of `{New}` to the parameter variable `{2}`, which is the second parameter the user entered to start the script. Notice that `{2}` is also placed inside double quotation marks, because its value is a text string that might contain spaces or special characters.

7. **Shift the parameters so that the next commands can work with the file or directory names the user entered.**

```
#   Shift parameters by 2 so that parameters are now all
#   file names or directories.
    Shift 2
```

The `Shift` command renumbers the parameters, subtracting 2 from each value. In other words, parameter `{3}` becomes parameter `{1}`, parameter `{4}` becomes parameter `{2}`, and so on.

8. **Open a control structure that loops through the parameters that are numbered above `{2}`.**

```
#   Loop through remaining parameters.
    For item in {parameters}
```

This command opens a `For … End` control structure that defines a variable named `{item}` that will contain each value in `{parameters}`, one by one. The variable `{parameters}` now contains each parameter the user entered, except the first two.

You can now add commands to the control structure that process the parameters.

9. **Add commands that check each parameter to make sure it is a valid file or directory name. If a parameter is not valid, display an error message and continue.**

```
        #   Make sure each parameter is a valid file or directory.
        #   If an invalid file or directory was given, write an
        #   error message and continue processing parameters.

            If !"`Exists "{item}"`"
                Echo "### {0}: {item} not a valid file or ∂
                    directory name."
                Continue
            End >> Dev:StdErr
```

These commands add an `If … End` control structure, which is nested within the `For … End` control structure. Within the control structure, the `Exists` command checks whether the file or directory name the user entered actually exists.

If the file or directory does not exist, the control structure displays an error message to standard error.

10. **Define a variable named `{files}`. If the user entered a filename, `{files}` will contain the full pathname of the file. If the user enters a directory name, `{files}` will contain the full pathnames of all files in the directory and its subdirectories.**

```
#   List the full path name if the parameter is a text file,
#   or list the full path names of all text files in the
#   directory and its subdirectories if the parameter is
#   a directory.  Discard diagnostic output in bit bucket.

    Set files "`Files -t TEXT -r -s -f "{item}" ≥ Dev:Null`"
```

This command sets the value of `{files}` to the output of the `Files` command. `Files` lists the pathname (or pathnames) of the file or directory contained in the variable `{item}`. The search is recursive, so that if `{item}` contains a directory name, `Files` lists the pathnames of all files in that directory and its subdirectories.

The `Files` command then discards any unwanted output by redirecting it to the virtual device `Dev:Null`.

11. **Open a control structure that loops through the files in the list.**

```
#   Loop through files in the list.
    For fName in {files}
    #   Make file the active file.
        Open "{fName}" ≥ Dev:Null

        Continue If "{Status}" != 0
```

These commands open a `For … End` control structure that defines a variable named `{fName}` that is one of the pathnames generated in step 10. The commands then open the file, discard any unwanted output, and continue to the next command if the file opens successfully.

Note that you must place `{fName}` inside double quotation marks, because a filename might contain spaces or special characters. You must also place `{Status}` inside double quotation marks, because a command might return a negative status code. If a command returns a status code such as –2, the – character is considered a special character.

12. **Add a command to the control structure that moves the insertion point to the top of the file.**

```
        #   Position cursor at the top of the file.
            Find • "{fName}"
```

Once the file is open, you can use the `Find` command to place the insertion point at the beginning of the file.

13. **Add a command to the control structure that replaces all instances of the selection with the new string.**

```
        #   Replace all instances of the selection with the ∂
        #   new string.
            Replace -c ∞ /{Old}/ "{New}" "{fName}"
```

The `Replace` command replaces every selection that matches the pattern stored in `{Old}` with the string stored in `{New}`. Note that you do not need to place the variable `{Old}` inside double quotation marks, because it is within forward slashes, which have the same effect. The forward slashes are part of the syntax of the `Replace` command.

Because this command is nested within a `For` … `End` control structure, the command is repeated for every filename that was entered on the command line.

14. **Add a command to the control structure that closes the file, saving any changes you have made. Then, close any control structures you have opened.**

```
        #   Close the file; save changes.
            Close -y "{fName}"
        End
    End
```

The `Close` command closes the file, saving the changes that have been made. The `-y` option causes the command to work without displaying any confirmation boxes, a feature that is useful when you write scripts.

You must then add an `End` command to close the control structure you opened in step 11, followed by a second `End` command to close the control structure you opened in step 8.

15. **Open the file SillyBalls.c and save it as SillyBalls.practice. Move SillyBalls.practice to the MPW directory.**

This step is important, because in the next step you will make a change that you do not want in the original SillyBalls.c file.

16. **To test the script, enter this command in the Worksheet window:**

```
ReplaceAll QuickDraw "QuickDraw MacGraw" SillyBalls.practice
```

Note that the second parameter in this command is placed within quotation marks, because it contains a space.

When you enter the command, the ReplaceAll script opens SillyBalls.practice, makes the changes, saves the file, and then closes it. If you reopen SillyBalls.practice, you can see the changes that have been made.

Note that the third line of code in SillyBalls.practice is now changed to

```
#include <QuickDraw MacGraw.h>
```

Oops! This is not the name of a valid include file.

17. **To reverse the changes, enter this command in the Worksheet window:**

```
ReplaceAll "QuickDraw MacGraw" QuickDraw SillyBalls.practice
```

The file is now just like the original.

Congratulations! You have now written an entire script. The complete script is shown in Listing 6-23.

Listing 6-23 The complete ReplaceAll script

```
#   ReplaceAll
#   MPW Shell Script
#   Copyright: 1993 by Apple Computer, Inc., all rights reserved.
#
#   Usage:
#       ReplaceAll selection newString name...

#   Function:
#       ReplaceAll replaces all instances of the selection with the
#       new string in the filename or filenames you specify. If you
#       specify a directory name, ReplaceAll works in all the files
#       in that directory and in all of its subdirectories.
#
#   -------------------------------------------------------------
#   Don't exit program on error.
```

```
    Set Exit 0
    Export Exit

#   Provide error detection.
    Set TraceFailures 1
    Export TraceFailures

#   Make sure there are at least 3 parameters.
    If "{#}" < 3
        Echo "### {0}: too few parameters"
        Echo "### Usage: {0} selection newstring files..."
        Exit 1
    End

#   Save pattern to find and replace in Old.
    Set Old "{1}"

#   Save string to replace it with in New.
    Set New "{2}"

#   Shift parameters by 2 so that parameters are now all file names
#   or directories.
    Shift 2

#   Loop through remaining parameters.
    For item in {parameters}

        #   Make sure each parameter is a valid file or directory.
        #   If an invalid file or directory was given, write an
        #   error message and continue processing parameters.
            If !"`Exists "{item}"`"
                Echo "### {0}: {item} not a valid file or ∂
                    directory name."
                Continue
            End >> Dev:StdErr

        #   List the full path name if the parameter is a text file,
        #   or list the full path names of all text files in the
        #   directory and its subdirectories if the parameter is a
        #   directory.  Discard diagnostic output in bit bucket.
            Set files "`Files -t TEXT -r -s -f "{item}" ≥ Dev:Null`"
```

```
#   Loop through files in the list.
    For fName in {files}
    #   Make file the active file.
        Open "{fName}" ≥ Dev:Null
        Continue If "{Status}" != 0
    #   Position cursor at the top of the file.
        Find • "{fName}"
    #   Replace all instances of the selection with the
    #   new string.
        Replace -c ∞ /{Old}/ "{New}" "{fName}"
    #   Close the file; save changes.
        Close -y "{fName}"
    End
End
```

Once you complete this tutorial, you can run the script to replace a selection with a specific text string in any number of files. For example, you might have a number of C source files that use a function named `paintOval`. You might then decide to change the name of the function to `paintCircle`—and wish to make the changes in a number of files stored in different directories.

In this tutorial, you have learned the proper procedures for

- beginning a script
- handling parameters the user enters
- initializing variables
- writing nested control structures that loop through each parameter and do the work of the script
- checking for errors
- closing a script

Appendixes

APPENDIX A

The Rules of Using Quotation Marks

This appendix summarizes the rules of using quotation marks that are presented in the chapter “Entering Commands.”

QUOTING RULE #1
Place quotation marks around parameters that contain spaces. ◆

QUOTING RULE #2
Always use a single or double quotation mark with a matching single or double quotation mark. ◆

QUOTING RULE #3
To use a special character as a literal character, enclose it in single quotation marks or place the ∂ character before it. ◆

QUOTING RULE #4
In a command, `∂n` inserts a return character, `∂t` inserts a tab character, and `∂f` inserts a form-feed character. ◆

QUOTING RULE #5
Use single quotation marks to enclose special characters used as literal characters. Do not use single quotation marks to enclose variables, special characters used as special characters, embedded commands, or command aliases. ◆

QUOTING RULE #6
To enclose variables, special characters used as special characters, embedded commands, and command aliases, use double quotation marks. ◆

QUOTING RULE #7
You can nest a pair of single quotation marks within a pair of double quotation marks, or a pair of double within a pair of single. To nest single within single, or double within double, place the ∂ character before each quotation mark in the inner pair. ◆

QUOTING RULE #8
Place variables within double quotation marks, unless they are defined as a list of individual items. ◆

QUOTING RULE #9
Mark quotation marks as literal characters with the ∂ character, or nest them within the opposite type of quotation marks. ◆

A P P E N D I X   B

Table of Special Characters and Operators

The MPW Shell special characters and operators are summarized in Table B-1. You can find complete descriptions of the special characters and operators in the appendixes of *MPW Command Reference*.

Table B-1 The MPW Shell special characters and operators

Character	Meaning	Keystroke
Arithmetic operators		
`+`	Addition	
`÷`	Division	Option-slash
`DIV`	Division	
`MOD`	Modulus	
`%`	Modulus	
`*`	Multiplication	
`-`	Subtraction, sign change	
Command terminators		
`||`	If not successful, do …	
`&&`	If successful, do…	
`;`	Join commands on one line	
Delimiters		
`"   "`	Enclose a literal string (with exceptions)	
`'   '`	Enclose a literal string (no exceptions)	
`{   }`	Enclose a variable	

Table B-1 The MPW Shell special characters and operators (continued)

Character	Meaning	Keystroke
{{ }}	Enclose a variable, saving quotation marks	
( )	Group commands or expressions	
` `	Pass output to another command	
`` ``	Pass output, saving quotation marks	
\ \	Search backward	
/ /	Search forward	
« »	Specify a repetition	Option-backslash, Option-Shift-backslash
[ ]	Specify a scan set	
Logical operators		
&	Bitwise AND	
~	Bitwise NOT	
|	Bitwise OR	
^	Bitwise XOR	
-=	Decrement a variable	
==	Equal to (strings, numbers, and variables)	
=~	Equivalent to (patterns)	
>	Greater than	
>=	Greater than or equal to	
≥	Greater than or equal to	Option-period
+=	Increment a variable	
<	Less than	
<=	Less than or equal to	
≤	Less than or equal to	Option-comma
&&	Logical AND	

Table B-1 The MPW Shell special characters and operators (continued)

Character	Meaning	Keystroke
`AND`	Logical `AND`	
`!`	Logical `NOT`	
`¬`	Logical `NOT`	Option-L
`NOT`	Logical `NOT`	
`OR`	Logical `OR`	
`||`	Logical `OR`	
`!=`	Not equal to (strings, numbers, and variables)	
`< >`	Not equal to (strings, numbers, and variables)	
`≠`	Not equal to (strings, numbers, and variables)	Option-equals
`!~`	Not equivalent to (patterns)	
`<<`	Shift left	
`>>`	Shift right	
Makefile characters		
`ƒ`	Depends on	Option-F
`ƒƒ`	Depends on, with own build commands	Option-F twice
Menu characters		
`-`	Add a separator to a menu	
`/`*c*	Assign the keystroke *c* to a menu item	
`(`	Disable a menu item	
`!`*c*	Mark a menu item with a character	
`^`*n*	Mark a menu item with an icon	
`<`*c*	Set the character style of a menu item	
Miscellaneous characters		
`∂`	Change the meaning of the next character	Option-D

Table B-1 The MPW Shell special characters and operators (continued)

Character	Meaning	Keystroke
#	Comment	
•	Separate parts of a filename	Option-8
:	Separate parts of a pathname	
…	Start `Commando`	Option-semicolon
Number prefixes		
`0b`	Binary number	
`$`	Hexadecimal number	
`0x`	Hexadecimal number	
`0`	Octal number	
Redirection characters		
|	Pipe output	
∑∑	Redirect all output and append	Option-W twice
∑	Redirect all output and replace	Option-W
≥≥	Redirect error messages and append	Option-period
≥	Redirect error messages and replace	
<	Redirect standard input	
>>	Redirect standard output and append	
>	Redirect standard output and replace	
Regular and selection expression characters		
•	Beginning of file, beginning of line	Option-8
§	Current selection	Option-6
∞	End of file, end of line	Option-5
:	Everything between two selections	
¡*n*	Move backward	Option-1
!*n*	Move forward	

Table B-1 The MPW Shell special characters and operators (continued)

Character	Meaning	Keystroke
¬	Not in the following list	Option-L
+	One or more occurrences	
∆	Place insertion point	Option-J
-	Specify a range of characters	
®	Tag an expression	Option-R
*	Zero or more occurrences	
Wildcards		
?	Match any character	
≈	Match any string	Option-X

A Sample UserStartup•*name* File

This appendix provides a sample UserStartup•*name* file that you can use to customize your MPW environment.

Note that some of the lines in this file are commented. These are lines that you can customize for your development environment.

```
#   UserStartup•mine
#   MPW Shell Script
#   Written by Gina Cherry, Mark Baumwell, and Godfrey DiGiorgi
#  August 16, 1991
#   Copyright: © 1991 by Apple Computer, Inc., all rights reserved.
#
#   Usage: UserStartup•mine is executed automatically from the Startup
#  script when MPW is launched.
#
#   Function: UserStartup•mine creates aliases, defines shell variables,
#  and adds frequently used commands to the menus.  This script is meant
#  to be an example of some of the commands you may want to include in your
#  UserStartup file.  It should be modified to conform to your file setup
#  and preferences.

#   Position cursor at end of worksheet.
    Find ∞ "{Worksheet}"

#   Define aliases
#  Aliases for shell commands.
    Alias       Cat     Catenate
    Alias       Cd      Directory
    Alias       Cp      Duplicate
    Alias       Grep    Search
    Alias       Ls      Files
    Alias       Mkdir Newfolder
    Alias       Mv      Move
    Alias       Pr      Print
    Alias       Pwd     Directory
    Alias       Rm      Delete
```

```
#   Aliases for applications.
#   Change these aliases to conform to your system.
    Alias       Res         '"HD:ResEdit"'
    Alias       MSWord      '"HD:Microsoft Word"'
    Alias       ALink       '"HD:AppleLink"'

# ---------------------------------------------------------------------------
#   Initialize shell variables.
# ---------------------------------------------------------------------------
#   Define a new shell variable Word.
    Set Word "[{WordSet}]+"
    #   Make Word variable available to other scripts.
        Export Word

#   Define a new shell variable Scripts.
    Set Scripts "{MPW}Scripts:"
    #   Make Scripts variable available to other scripts.
        Export Scripts

#   Define a new shell variable Source.
#   Change this command to conform to your system, and then uncomment it.
#   Set Source "{MPW}Source Code"
    #   Make Source variable available to other scripts.
    #   Uncomment this command also.
    #   Export Source

#   Define shell variable openList to be used by the script MyOpen.
#   Change this definition to conform to your system, and then uncomment it.
#   Set openList "':' '{Scripts}' '{Source}' '{MPW}'"
    #   Make openList variable available to other scripts.
    #   Uncomment this command also.
    #   Export openList

#   Set the Commands shell variable, which tells the shell where to
#   look for commands to execute. Change this definition to conform
#   to your system.
    Set Commands ":,{MPW}Tools:,{MPW}Scripts:,{MPW}MyTools:,{MPW}MyScripts:"
    #   Make the Commands variable available to other scripts.
        Export Commands
```

```
#   Set the DirectoryPath shell variable, which tells the shell where
#   to look for directories when changing directories. This allows the
#   user to type shorter pathnames when changing to commonly used directories.
#   Change this definition to conform to your system.
    Set DirectoryPath ":,{MPW},{Scripts},{MPW}MyScripts:,{Source}"
    #   Make DirectoryPath variable available to other scripts.
        Export DirectoryPath

#   Change the following definitions to suit your preferences:

    #   Set the default font for new windows.
        Set Font Geneva

    #   Set the default font size for new windows.
        Set FontSize 10

    #   Set the default window size for new windows.
        Set NewWindowRect 0,0,400,50

    #   Set default window size for zooming.  In this example, ZoomWindowRect
   #  is set to a reasonable size for a 2 page monitor, so that windows
   #  won't zoom to the full size of the monitor when the user clicks
   #  on the zoom box.
        Set ZoomWindowRect 0,550,825,1100

    #  Set options used by print menu.  Print page headers at top of each
    #   page. Set form feed to œ (Option-Q). Print line numbers to left
    #   of printed text.  Write progress information to diagnostic output.
        Set PrintOptions '-h -ff œ -n -p'

# -------------------------------------------------------------------------
#   Mount Current Projects
# -------------------------------------------------------------------------
#   Change this command to mount projects on which you are currently working,
#   and then uncomment it.
#   MountProject MyProject

# -----------------------------------------------------------------------
#   Customize menus
# -----------------------------------------------------------------------
# Modify the Edit menu.
```

```
    #   Add disabled separator bar.
        AddMenu Edit '(-' ''
    #   Add commands and their keyboard equivalents.
        AddMenu Edit 'Show Invisibles/2' 'ShowNPChar'
        AddMenu Edit 'Comment/3' 'Comment "{Active}"'
        AddMenu Edit 'LtoU/4' 'LtoU "{Active}"'
        AddMenu Edit 'UtoL/5' 'UtoL "{Active}"'

#   Modify the Find menu.
    #   Add Commando dialog for ReplaceAll
        AddMenu Find 'Replace All/®' 'ReplaceAll…'
#   Add disabled separator bar.
        AddMenu Find (- ''
    #   Add GoTo command to the Find menu.
        AddMenu Find 'GoTo/7' 'GoTo "{Active}"'
    #   Add Commando dialogs for WhereIs and Search to the Find menu.
        AddMenu Find 'WhereIs/8' 'WhereIs…'
        AddMenu Find 'Search/9' 'SearchAll…'
        AddMenu Find 'Search & Set/¥' 'SS'

#   Create the Directory menu.
#   Change this command to add other frequently used directories to the
#   parameter list, and then uncomment it.
#   DirectoryMenu "{Scripts}"  "{Source}" `Directory`

#   Modify the Projector menu
    #   Add disabled separator bar.
        AddMenu Project '(-' ''
    #   Add Commando dialog for NameRevisions command.
        AddMenu Project 'Name Revisions…' 'NameRevisions…'
    #   Add command for RevertToVersion script.
        AddMenu Project 'Revert To Old Version…' ∂
            'Begin; ∂
                Set _name_ `Request "Filename to revert?"`; ∂
                If "{_name_}"; ∂
                    RevertToVersion "{_name_}"; ∂
                End; ∂
                Unset _name_; ∂
            End ∑∑ "{worksheet}"'
```

```
# Create the Tools menu
    #   Add disabled separator bar and header.
        AddMenu Tools '(-' ''
        AddMenu Tools '(Application Tools  ' '
        AddMenu Tools '(-' ''
    #   Add applications to Tools menu
    #   Change this section to add other frequently used applications.
        AddMenu Tools 'Microsoft Word' 'MSWord'
        AddMenu Tools 'AppleLink' 'ALink'
        AddMenu Tools 'ResEdit' 'Res'

    #   Add disabled separator bar and header.
        AddMenu Tools '(-' ''
        AddMenu Tools '(Other Tools  ' ''
        AddMenu Tools '(-' ''
    #   Add other tools to Tools menu.
        AddMenu Tools 'Prototype C Functions' 'ProtoGen…'
        AddMenu Tools 'Mark C Functions' 'MarkIt…'
        AddMenu Tools A2Hex     'A2Hex "{Active}" ∑∑ "{Worksheet}"'
        AddMenu Tools 'Conversion Table' 'Open -r ∂
            "{MPW}Scripts:Dec-Oct-Hex-BinTable"'
        AddMenu Tools 'Indicate 80 Columns ' 'Indicate80Columns'

    #   Create the Commands menu
    #   Add disabled separator bar and header.
        AddMenu Commands '(-' ''
        AddMenu Commands '(ACTIVE' ''
        AddMenu Commands '(-' ''
    #   Add command to echo the name of the active window to the Worksheet.
        AddMenu Commands 'Echo Active' 'Echo "{Active}" >> {Worksheet}'
    #   Add command to position cursor at the top of the active window.
        AddMenu Commands 'TOP of Active file' 'Find • "{Active}"'
    #   Add command to position cursor at the bottom of the active window.
        AddMenu Commands 'BOTTOM of Active file' 'Find ∞ "{Active}"'

    #   Add disabled separator bar and header.
        AddMenu Commands '(-' ''
        AddMenu Commands '(TARGET' ''
        AddMenu Commands '(-' ''
    #   Add command to echo the name of the target window to the Worksheet.
        AddMenu Commands 'Echo Target' ∂
```

```
            'Echo "{Target}" >> "{Worksheet}"'
    #   Add command to position cursor at the top of the target window.
        AddMenu Commands 'TOP of Target file' 'Find • "{Target}"'
    #   Add command to position cursor at the bottom of the target window.
        AddMenu Commands 'BOTTOM of Target file' 'Find ∞ "{Target}"'

    #   Add disabled separator bar and header
        AddMenu Commands '(-' ''
        AddMenu Commands '(Window Management' ''
        AddMenu Commands '(-' ''
    #   Add command to save all windows.
        AddMenu Commands 'Save All/E' "Save -a"
    #   Add command to close all windows.
        AddMenu Commands 'Close All/K' "Close -a"
    #   Add disabled separator bar.
        AddMenu Commands '(-' ''
        AddMenu Commands 'RotateWindows/®' 'RotateWindows'
        AddMenu Commands 'Cleanup Windows/©' "CleanWindows"
        AddMenu Commands 'Side-By-Side/ß' "VTileWindows ≥ Dev:Null"
        AddMenu Commands 'Small Window/∑' "SmallWindow"

        AddMenu Commands '(-' ''
        AddMenu Commands '(Other Commands' ''
        AddMenu Commands '(-' ''
        Execute AddDeleteMenu
        AddMenu Commands 'Search and Open/Ω' 'MyOpen'

#   Set the current directory to Source.
#   Change this command to set directory to the directory of
#   your preference, and then uncomment it.
#   SetDirectory "{Source}"
```

Glossary

active window The window that is currently selected. If you have many windows stacked from front to back, the active window is the frontmost window.

alert box A box that displays a message to your user. Alert boxes can have one or more buttons, and the user is required to click a button before the box disappears.

alias See **command alias, directory alias**, **embedded alias, Finder alias, leaf alias.**

built-in command A command that is executed by code that resides in the MPW Shell.

command alias An alternate name for an MPW Shell command that you define with the `Alias` command.

command error An error that occurs in a script when a command does not work as you expect it to, for example, when a command tries to open a file that does not exist.

command interpreter The part of the MPW Shell that reads and executes commands you enter.

command name The first word of a command line, which may be followed by options and parameters.

Commando The tool that displays Commando dialog boxes.

Commando dialog box A dialog box that allows you to execute any MPW Shell command interactively, by clicking on radio buttons and checkboxes, choosing from pop-up menus, and typing in text boxes.

command terminator A character that ends a command and, in some cases, introduces another command.

control structure A group of commands that are not executed sequentially, as commands ordinarily are, but that create loops and iterations.

current directory The directory that is presently the default directory. You set the current directory with the Directory menu or the `Directory` command.

current selection The text that is presently selected and highlighted in a file.

data fork The part of a file that contains data and that is accessed through the File Manager.

development environment The collection of tools and resources you need to develop software.

diagnostic output The data stream a command uses for its error messages. By default, MPW Shell commands display diagnostic output in the active window.

directory A holder for files, applications, and other directories. In this guide, the same as a folder.

directory alias A Finder alias whose target is a folder or directory.

double backquote In the MPW Shell, the `` character. The `` character is placed on either side of a command whose output is used as the parameter to another command, when the output contains single or double quotation marks.

editing Making changes to a text file either by using windows, menus, and dialog boxes or by entering commands.

editor The part of the MPW Shell that you use to create and edit text files using windows, menus, and dialog boxes.

embedded alias A Finder alias that appears at the beginning or in the middle of a pathname.

embedded command A command whose output is used as a parameter to another command. Embedded commands are placed within backquotes.

enclosing command A command that takes an embedded command as a parameter.

enclosing script A script that calls another script. The script that is called is a nested script.

export To extend the scope of a variable, so that the variable is recognized in contexts other than the one in which it is defined.

expression A formula in a command that defines a calculation to be performed or some text that a command should select. See also **selection expression** and **regular expression**.

file alias A Finder alias whose target is a file or application.

Finder alias An alias for a file, directory, or volume that you create from the Finder.

folder alias See **directory alias.**

full pathname The complete name that locates a file, from the volume name, through any intermediary directory names, to the filename.

grandparent directory The directory two levels above the current directory.

hierarchical file system (HFS) The directory structure in which files are stored on the Macintosh.

inactive window A window that is open but not presently selected.

leaf alias A Finder alias that occurs at the end of a pathname.

leafname A special type of partial pathname that contains just a filename, with no colons.

makefile A file that contains the commands and dependency rules that are necessary to build a source file into an executable application.

markers Tags that you can insert into a text file so that you can locate sections of it easily.

nested script A script that is opened by another script. The script that opens a nested script is called an enclosing script.

nonprinting characters Invisible characters that may have been inserted into a text file.

operand A value in an expression that is compared or combined with another value.

operator A special character or combination of special characters that compares or combines two values in an expression.

option A part of a command that is always preceded with a hyphen (such as `-b` or `-v`) and indicates how a command is to be carried out.

parameter A part of a command that specifies something the command should act on, such as a filename or a window name.

parameter variable A variable that is assigned to a parameter on the command line a user enters to start a script.

parent directory The directory one level above the current directory.

partial pathname (1) A pathname that locates a file, starting partway down the path. (2) A pathname that starts at a volume name but ends before a filename.

pathname A name that describes the route one would take to locate a file in the Macintosh hierarchical file system (HFS).

regular expression A combination of text and special characters that specify a pattern that might be matched by more than one string.

resource fork The part of a file that contains data an application uses, such as fonts, menus, and icons. An executable file's code is also stored in the resource fork.

scan set A type of regular expression consisting of one or more characters enclosed in brackets. A scan set can match any character in the list, any character not in the list, or any of a range of characters.

script A sequence of commands stored in a file that can be executed by typing the name of the file.

selection A series of characters, or a character position, at which the next editing operation will occur. Selected characters in the active window are inversely highlighted.

selection expression A formula in an editing command that defines the text to be selected.

shell A utility that accepts your commands, interprets them, and passes them on the appropriate program for execution.

Shell variable A variable that you define and use only in MPW Shell commands and scripts. Shell variables are not used in any of MPW's programming languages.

special character A nonalphabetic character that has a special meaning in a command.

split bar A small black rectangle in horizontal and vertical scroll bars of a window that allows you to divide the window into panes.

standard error The data stream a command uses for its error messages. By default, the MPW Shell directs error messages to the active window.

standard input The data stream from which a command takes its input. By default, the MPW Shell accepts as standard input the characters you type from your keyboard.

standard output The data stream used for output from a command. By default, the MPW Shell directs standard output to the active window.

stationery pad A file that serves as a template from which you can create other documents. When you create a document from a stationery pad, the stationery pad remains unchanged, so that you can use it again.

status panel The small rectangle in the upper left of the active window that shows which command is presently being executed.

syntax error An error that occurs in a script when a command does not have the proper syntax; for example, if a required quotation mark has been left out.

target The original file, directory, or volume from which a Finder alias was created.

target window The window that was last active. The target window is where editing commands usually take effect.

tool A small executable program stored in the Tools directory that you can execute by entering its name as a command in the active window.

usage error An error that occurs in a script when the command line the script's user enters to start the command is incorrect.

virtual device A source of input or a destination for output that does not have a corresponding physical device.

volume A hard disk or floppy disk that can be formatted to store files.

wildcard A special character that can match a number of other characters.

Worksheet window A window the MPW Shell displays while it is running. You type commands and receive information from the MPW Shell in the Worksheet window because the Worksheet window is always open as you use the MPW Shell.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Suzanne Ahmed, Daphne Steck

DEVELOPMENTAL EDITOR
George Truett

ILLUSTRATOR
Ruth E. Anderson, Sandee Karr

PRODUCTION EDITOR
Rex Wolf, Alex Solinski

Special thanks to Greg Branche, Joanna Bujes, Ray Chiang, Mark Cleveland, Lee Collings, Rick Dorin, Rich Fleishman, Herb Kanner, Judy Kettefhofen, Gary Little, Peri Frantz, Pete Richardson