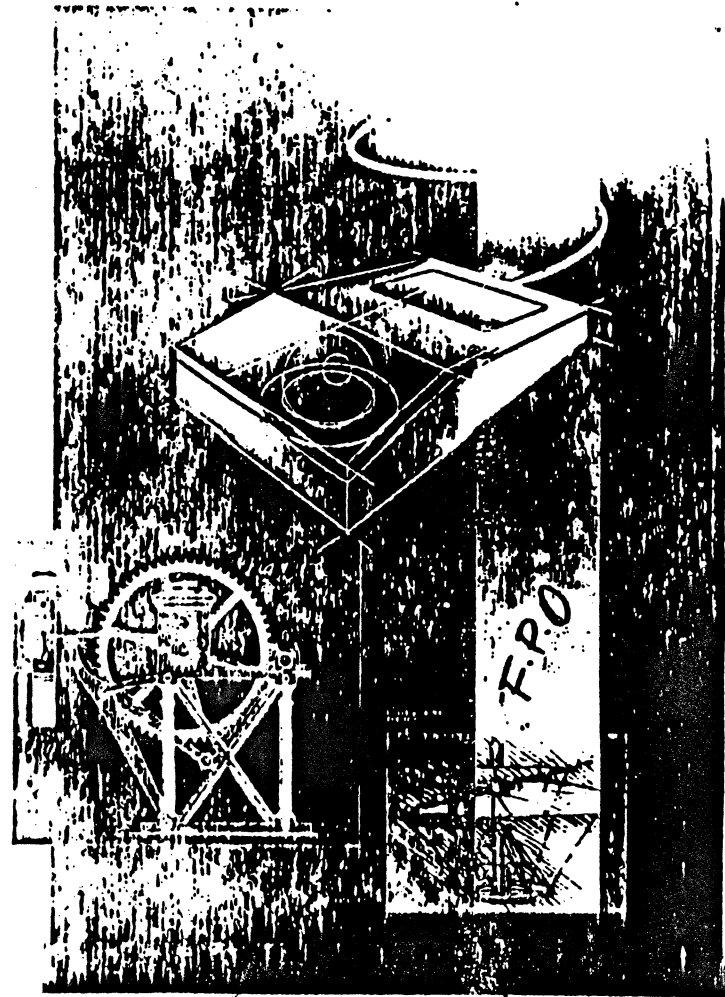# MPW: Macintosh Programmer's Workshop Development Environment, Volume 1

Version 3.0

Macintosh®

Macintosh Programmer's Workshop 3.0 Reference

Contents

4   Using MPW: The Basics   87

6 Advanced Editing 171

7 Projector: Project Management 195

12   Writing an MPW Tool   341

G   The Graf3D Library   547

Figures and tables

14 Performance-Measurement Tools 447

B Summary of Selections and Regular Expressions 495

C Special Operators 501

E File Types, Creators, and Suffixes 519

F Tool Libraries 523

Part I **Shell Reference**

Introduction: The New and the Necessary

WELCOME TO THE MACINTOSH® PROGRAMMER'S WORKSHOP 3.0. This introduction is your guide to the new features and enhanced capabilities.

Those currently using MPW™ 2.0 are urged to carefully review the section "What's New in MPW 3.0" because many changes may affect your MPW 2.0 scripts and other ways of doing things. The last two sections of this introduction describe new hardware and software requirements as well as revised notation conventions and reorganized documentation. If you are new to MPW you can skip the "What's New in MPW 3.0" section, but be sure to read "What You'll Need" and "About This Reference." This last section guides you to the parts of this book that help you get started. ■

Contents

Power tools for Macintosh programmers

The Macintosh Programmer's Workshop (MPW) provides professional software development tools for the Apple® Macintosh computer. Briefly, MPW 3.0 consists of the following parts:

- MPW Shell (the programming environment)
- Project management system (Projector Trademark)
- Resource compiler and decompiler (Rez and DeRez)
- Resource editor (ResEdit™)
- Linker (Link)
- Make (for tracking file dependencies)
- Dialog interface (Commando)
- Symbolic Application Debugging Environment (SADE™, an interactive symbolic debugger) and MacsBug
- Performance-measurement tools

Note that ResEdit, although still part of MPW, has been enhanced and is now documented separately. Also, the new interactive debugger, SADE, and an improved MacsBug are now each documented in their own separate reference works, included with the MPW product.

The system also includes a comprehensive array of additional tools for creating and manipulating text and resource files. The following MPW products are separately available:

- **Macintosh Programmer's Workshop 3.0 Assembler** provides everything you need to develop applications, tools, and desk accessories in assembly language, including the ability to create macro libraries.

- **Macintosh Programmer's Workshop 3.0 Pascal** provides the additional tools, interfaces, and libraries you need to develop applications, tools, and desk accessories in Pascal.

- **Macintosh Programmer's Workshop 3.0 C** provides a new C compiler and a C++ translator along with the interfaces and libraries needed to develop applications, tools, and desk accessories in C or C++.

- **MacApp®**, the Expandable Macintosh Application, provides of a set of object-oriented libraries that automatically implement the standard Macintosh user interface, thus simplifying and speeding up the process of software development. Either MPW Pascal or MPW C++ is required for use of MacApp.

The entire MPW system is outlined in detail in Chapter 1, "System Overview."

The Macintosh Programmer's Workshop 3.0 provides these advantages over previous development systems:

- **Integration:** The numerous utilities and tools of the MPW system all run within the MPW Shell environment. The integrated environment enables separately developed applications, called **MPW tools**, to run within the programming environment. The MPW editor is always available to generate both text and command lines; there is no distinction between command and text windows.

- **MultiFinder™ compatibility:** MPW 3.0 tools can now be operated in the background when using Macintosh System 6.02 with MultiFinder. This means that you can switch to another application while a tool, such as a compiler, is running. You can also configure your system so that you can use the MPW Shell for editing or other operations while a tool runs in the background. See "Using MPW With MultiFinder" in Chapter 2.

- **Project management:** Projector, a new program integrated with MPW, makes it easy to keep track of large projects involving many programmers, or simply to maintain an orderly revision history, showing who did what to every file and why. You can use Projector to **branch**, that is, create many experimental versions of a file at any stage in its evolution—without risk of confusion.

- **Automated build process:** A pull-down menu provides several ways to build or rebuild your programs quickly and automatically. You can also automate complex builds by using the Make tool and command-language scripts.

- **Command scripting:** In addition to menu commands, MPW provides a full command language, including Shell variables, command aliases, pipes, and the ability to redirect input and output. You can combine any series of commands into an elaborate, specialized script (command file) for fast, accurate, automatic results.

- **Regular expression processing:** The editor component of the Shell provides powerful search and replace capabilities with regular expressions, which form a language for describing complex text patterns. Regular expressions allow you, for instance, to restructure complex tables with a single command.

- **Extensibility:** You can customize MPW in just about any way you can imagine. You can create your own integrated tools and scripts to run within the Shell environment. You can also add your own menus, menu items, and dialogs to the Shell.

- **Ease of use:** On-line help is available at all times. In addition, the Commando dialog interface gives you immediate on-screen access to all of MPW's versatile options and functions in specialized dialog boxes. This interface makes learning easier and faster. You can compose complex command lines without referring to the manual. And you can create a Commando interface for your own tools and scripts as well.

MPW 3.0 provides a customizable programming environment with a completeness, power, and flexibility unmatched by any other Macintosh-based system. Because it is full-featured and extremely versatile, the first-time user should be prepared to devote some time to learn it. This effort will be well repaid by the power and versatility that MPW places in your hands.

What's new in MPW 3.0

MPW 3.0 is faster and easier to use than its predecessor and now fully exploits MultiFinder. Use of MPW with MultiFinder greatly increases its convenience and efficiency. Many new tools and options to existing tools have been added. Major additions to MPW include Projector, a project management system, and SADE, the Symbolic Application Debugging Environment. MPW now also supports C++. These innovations are each briefly described in the sections that follow. Changes to menus, tools, variables, and compilers are itemized in the lists that follow.

If you are currently using MPW 2.0, it is especially important that you carefully review the changes listed in this section. The extensive changes implemented in MPW 3.0 might affect scripts written for the MPW 2.0 Shell.

MPW C++

MPW now includes extensions to the C language that support the features of C++. MPW C++ is an approximate superset of the C programming language that maintains the efficiency and power of C while adding features such as operator overloading (which lets you define additional meanings for built-in operators), ANSI-like type-checking, automatic type conversion, and class hierarchies with inheritance. Because C++ supports object-oriented programming, developers who prefer C to Pascal can now take advantage of MacApp. For more information, see *MPW 3.0 C++ Reference.*

Projector

MPW 3.0 includes an easy-to-use, built-in project management system, Projector, that can be customized to fit any working style, from that of the single programmer to that of the large, networked engineering team. Briefly, here's how it works: You check out a file or group of files from Projector for either review or modification. Although many people can review a file, only one person at a time can modify it. When you've finished your work, you check the file back in with Projector, along with a note detailing your modifications. Your name, your notes, and the date are automatically filed in Projector's revision history for that project. It's also possible to create parallel branches of a single project for experimental purposes. Chapter 7 is a detailed account of Projector.

Symbolic Application Debugging Environment (SADE)

The Symbolic Application Debugging Environment (SADE) allows you to monitor the execution of a program at both the processor level and at the symbolic program source level. Both SADE and the enhanced MacsBug are now each documented separately from the *MPW Reference*. For more information, see the *MacsBug Reference* and the *SADE Reference*.

New or enhanced tools

The tools and scripts included with MPW 3.0 have been improved in many ways for increased versatility. These enhancements are briefly catalogued in the list that follows. In addition, a number of new tools and scripts have been added to support Projector and the C++ compiler.

MPW 3.0 supports shared tools on a network file server.

These rarely used conversion tools are no longer included with MPW but are still available from Technical Support at Apple Computer, Inc.:

- TLACvt
- MDSCvt
- CVTObj

All MPW 3.0 commands, including tools and scripts, are individually documented in the alphabetically organized Command Reference in Part II of this book; that is the first place to look for more information about any tool.

- **Backup:** Two new options have been added.

- **C:** The C compiler has been completely rewritten for MPW 3.0. Some of the options and calling conventions are different from those in the MPW 2.0 C compiler. See the *MPW 3.0 C Reference.*

- **CFront:** New translator for C++.

- **Choose:** A new tool that enables you to mount servers and select Apple Laserwriter® printers from within the MPW environment.

- **Commando:** Now has a built-in editor that makes it easy to modify Commando dialog boxes.

- **CompareFiles:** A script that compares two files side by side, pinpointing any differences.

- **CompareRevisions:** A Projector script used to identify and compare revisions. See Chapter 7 for details.

- **CPlus:** New script that compiles C++ programs.

- **CreateMake:** Enhanced with a new option that supports SADE.

- **DoIt:** A script to highlight and execute a series of commands.

- **DumpCode:** Enhanced.

- **DumpFile:** New -bf option. Note that the -c option has been renamed -w (for width).

- **DumpObj:** Enhanced to support SADE. Two new options have been added.

- **GetFileName:** Enhanced with a new -c (current) option to write the current Standard File pathname to standard output. The syntax of this command has also been improved.

- **GetListItem:** GetListItem now supports keyboard shortcuts and a new option: -s (single) option that permits only a single item to be selected from a displayed list.

- **Lib:** Enhanced. Lib now determines the optimum buffer allocation from the amount of available memory; the old -b, -bs, -bf options are therefore obsolete and have been eliminated.

- **Link:** Enhanced to permit up to 1024 files, including both object files and symbolic debugger source file specifications. A new -map option produces a sophisticated link map. Link now determines the optimum buffer allocation from the amount of available memory; the old -b, -bs, -bf options are therefore obsolete and have been eliminated. A new option supports SADE. See Chapter 10.

- **MacsBug:** MacsBug performance has been enhanced and upgraded. The MC68881 and MC68882 floating-point coprocessors are supported. See the separate *MacsBug Reference.*

- **Make:** Changes have been made to the way variables are treated. See "Variables in Makefiles" in Chapter 9.

- **MatchIt:** A new command that intelligently seeks the mate of a specified delimiter used in Pascal, C, or Assembler, allowing for loops, comment fields, nesting, and so on.

- **MergeBranch:** A Projector script used to help merge a branch file back into the trunk of a project. See Chapter 7 for details.

- **Pascal:** Enhanced with object-oriented capabilities. See the MPW Pascal section later in this chapter and the *MPW 3.0 Pascal Reference.*

- **Print:** Enhanced. A new option, **-ps**, lets you send a file of PostScript® commands to the LaserWriter™.

- **ProcNames:** This Pascal utility now generates Shell marker commands, allowing easy access to the procedure, function headers, or bodies. Names are now displayed indented to show their nesting level. Nesting level and line number are also displayed.

- **Resource tools:** The command language of Rez has been extended with the new syntax element Label to support color QuickDraw resources. There are a few new syntax rules, new options, and two new functions that allow you to delete resources or change resource information. See Chapter 11 and Appendix D.

- **Sort:** A new tool for sorting lines of text.

- **WhereIs:** This new tool helps you find files hidden deep in a directory tree. You can use it to locate files when you know only a partial pathname.

New or enhanced Shell commands

All of these built-in commands are fully described in the Command Reference in Part II; that is the first place to look for more information.

- **CheckIn:** New Projector command to check files in to a Project. See Chapter 7.

- **CheckOut:** New Projector command to check files out from a Project. See Chapter 7.

- **CheckOutDir:** New Projector command to set Checkout directory. See Chapter 7.

- **Close:** Enhanced with a **-c** option; it lets you select the dialog's Cancel button during a scripted operation.

- **Date:** Enhanced to provide "date arithmetic."

- **DeleteNames:** New Projector command. See Chapter 7.

- **DeleteRevisions:** New Projector command. See Chapter 7.

- **Directory:** A "directory path" variable (similar to the {Commands} variable) for changing current directories has been added.

- **Evaluate:** Enhanced to support different radices and variable assignments.

- **Flush:** A command for flushing tools from the tool cache.

- **Format:** A scriptable form of the format option in the Edit menu.

- **FullBuild:** Enhanced.

- **ModifyReadOnly:** New Projector command to make read-only files modifiable. See Chapter 7.

- **MountProject:** New Projector command. See Chapter 7.

- **MoveWindow:** Enhanced to provide current window size and position.

- **NameRevisions:** New Projector command to name revised projects. See Chapter 7.

- **NewProject:** New Projector command to create a new project. See Chapter 7.

- **OrphanFiles:** New Projector command. See Chapter 7.

- **Position:** This new command shows the current line number, beginning of selection, and end of selection in specified windows.

- **Project:** New Projector command. See Chapter 7.

- **ProjectInfo:** New Projector command. See Chapter 7.

- **Request:** Enhanced with -q option to quiet any error messages, permitting a script to continue regardless of user input.

- **RotateWindows:** New command that sends the front window to the back.

- **SizeWindow:** Enhanced to provide current window size and position.

- **StackWindows:** Enhanced to support user-defined rectangles and a variable number of windows.

- **TileWindows:** Enhanced to support user-defined rectangles and a variable number of windows.

- **TransferCkid:** New Projector command. See Chapter 7.

- **UmountProject:** New Projector command. See Chapter 7.

New Shell editor capabilities

The MPW Shell editor has been refined in various ways:

- MPW 3.0 supports the special keys on the Apple Extended Keyboard:

  Esc   Same as Cancel button in a dialog box

  Undo   Same as Undo menu command

  Cut   Same as Cut menu command

  Copy   Same as Copy menu command

  Paste   Same as Paste menu command

  Help   With no selection, displays a summary of the Help available. With a
            selection, information on that selection is displayed.

  Home   Equivalent to moving the vertical scroll box to the top of the scroll bar.

  End   Equivalent to moving the vertical scroll box to the bottom of the scroll
            bar.

  Page Up   Equivalent to clicking the mouse pointer in the upper gray region of the
            vertical scroll bar.

  Page Down Equivalent to clicking the mouse pointer in the lower gray region of the
            vertical scroll bar.

- The displayed line-length limit has been increased to 256 characters.

- The tab-length limit has been increased to 100 characters.

- Horizontal scrolling is faster; more screen area is moved per mouse click.

- You can reverse the direction of the Find, Find and Replace, Find Same, Replace Same,
  and Find Selection functions by holding down the Shift key when selecting a menu
  item (or, in a dialog box, when clicking OK). This makes interactive searching a little
  more convenient but does not affect Shell search variables.

- Text selection by matching delimiters (such as { }, ( ), [ ], and so on), has been
  modified. Instead of selecting the rest of the document when a matching character is
  not found, the delimiter at the position of the double-click is highlighted. During the
  search you can abort by pressing Command-Period.

- The new commands Format and Position (described above in the "New or Enhanced
  Shell Commands" section) are useful for scripted editing.

- The library routine `faccess` has been enhanced to provide more programmatic
  control over Shell windows.

- You can now disable Auto-Indent for one line by pressing Option-Return.

- The MPW Shell editor ignores any zero-width characters that are typed from the keyboard. (Usually these are typed by accident.) If you really want a control character in your document, you can enter it in the Key Caps desk accessory and then paste it in your document. To delete control characters that might not be visible, select Show Invisibles from the Format dialog box. All control characters are displayed as an inverse question mark (¿).

New standard Shell variables

Twelve new variables have been added to give you control over almost all formatting and editing options from scripts. (Only display invisibles cannot be predefined.) The first five variables listed here provide default settings for new windows and are especially useful with large-screen monitors. See "Variables Defined in the Startup File" in Chapter 5 for more information.

- {AutoIndent} sets default indenting for new windows.

- {Font} sets default font for new windows.

- {FontSize} sets default font size for new windows.

- {NewWindowRect} sets the default size for new windows.

- {ZoomWindowRect} sets default size for windows that are zoomed to full screen size.

- {TileOptions} sets options for the TileWindows menu item, for example, to specify a rectangle for the tiled window arrangement.

- {StackOptions} sets options for the StackWindows menu item, for example, to specify a rectangle for the stacked window arrangement.

- {SearchBackward} can be used to set your default environment to specify backward searching.

- {SearchType} can be used to set your default environment to specify searching for literal characters, words, or regular expressions.

- {SearchWrap} can be used to set your default environment to specify wrap-around searching.

- {User} specifies the name of the user currently using MPW. It is predefined to be the same as the user name specified in the Chooser.

- {IgnoreCmdPeriod} is a new variable referenced by MPW's command interpreter. Use this variable in your scripts when you want any Command-Period input by the user to be ignored.

Changes to menus and dialogs

A few menus have been slightly changed since the release of MPW 2.0:

- **TileWindows and StackWindows menu items** now, by default, do not include the Worksheet. You can include the Worksheet in the tiling or stacking by pressing the Option key when selecting the TileWindows or StackWindows menu item. The {TileOptions} and {StackOptions} variables let you completely customize the operations of the TileWindows and StackWindows menu commands. See Chapter 3.

- **Window menu** now lists any open Projector windows. See Chapter 3.

- **The Open dialog box** now contains a Read Only checkbox.

Miscellaneous Shell changes

Here are some important improvements for the MPW Shell:

- MPW 3.0 supports background operation of tools while running MultiFinder. This is a significant improvement in convenience and efficiency. Please see "Using MPW With MultiFinder" in Chapter 2 for instructions on configuring your system for true multitasking.

- An automatic installation program is included with MPW 3.0. This program, Installer, and the tools to support it, can be found on the MPW Installation Disk. Please read "Installing the System" in Chapter 2 before doing anything with it. This is important because the arrangment of MPW files on the 3.5-inch distribution disks has been changed to represent their final destination when moved to a hard disk. Thus there will be some duplication of folders across the set of distribution disks so that you cannot simply copy the entire contents of a distribution disk without some conflict.

- The Startup file now executes UserStartup and then any file named UserStartup•*name* in the directory that contains the Shell. (Press Option-8 to obtain the • symbol.) If you have a customized UserStartup file, you may want to personalize it (for example, UserStartup•Tom) so that when you install MPW 3.0 your customized file won't be overwritten.

- Standard output and diagnostic output can now be directed to the same place with the Σ (Option-W) character, meaning: "The summation of all output..." See "Redirecting Input and Output" in Chapter 5 for the new syntax.

- You can now use Option-Enter to invoke the Commando dialog boxes for commands. Alternatively, you can still type the command name, then the ellipsis character (Option-Semicolon), and then press Enter.

- A new directory path variable for changing current directories is now available from the Directory command. (See Part II.)

- Numeric variables have been added to the Shell command language. See the Evaluate command in Part II for details.

- The notation conventions of this reference have been slightly modified. The index has also been improved. See "About This Reference" at the end of this Introduction.

Numeric libraries

Linking with numeric libraries has been simplified by placing certain conversion functions, such as `num2dec`, in CRuntime.o. A program that simply uses `printf` will no longer need to link with CSANElib.o.

A new {AIncludes} macro file called SANEMacs881.a is provided as a migration aid for Macintosh II developers who seek even greater floating-point performance from their products by using SANE macros. With little modification of their source files, they can reassemble by using the 881 SANE macros and thereby generate a faster application that runs only on the Macintosh II.

MPW C and MPW C++ Include files

The capitalization conventions for those functions that use Points or strings have been changed for MPW 3.0. These changes are itemized here:

- Those functions that call "glue" code to convert C strings to Pascal strings or dereference Points are now spelled with all lower case letters.

- The in-line versions of those function calls, those that do no conversions, are now spelled with mixed cases to match the conventions in *Inside Macintosh.*

- You will find in the Scripts folder a new script, CCvt, that changes source code to conform to the new standard spelling conventions. CCvt first backs up the original source and then uses two Canon dictionaries to change mixed case spellings to all lower case and all upper case spellings to mixed case.

- The syntax for ROM calls (A-traps) has been changed. The new syntax allows multiple instructions for "direct functions" and is more compatible with standard ANSI C and C++.

- The header files have been rewritten with function prototypes that allow ANSI C and C++ to do additional type-checking and code optimization.

If you use MPW C, please see the *MPW 3.0 C Reference* for more information about interfaces.

MPW Pascal

The MPW 3.0 Pascal Compiler no longer provides the compiler directive $LOAD and the option -z that were supported in MPW 2.0 Pascal. In addition to providing nearly all the capabilities described in the ANS Pascal Standard, MPW 3.0 Pascal expands the power and flexibility of Pascal programming with a range of new features and options:

- SADE, the symbolic debugger (-sym option), and MacsBug (-mbg option) are supported.

- A replacement for the $LOAD mechanism provides a more automatic and faster method (-noload, -clean, and -rebuild options).

- You can use character constants as valid string expressions.

- Symbol support for MacsBug has been extended and improved.

- Global data greater than 32K is now possible.

- The requirements for forward type references are more flexible.

MPW tool libraries

MPW language libraries that control the MPW Shell were previously documented in their respective language references. All Shell-related routines are now combined in this reference.

- Use of the MPW cursor control routines and error file manager is now explained in Appendix F of this book. Examples are shown in both MPW C and MPW Pascal; Assembly programmers can use both.

- Use of the MPW Integrated Environment routines are documented in Chapter 12. The routines are explained for MPW Assembler, MPW C, and MPW Pascal.

- The Graf3D library is now documented in Appendix G. Each routine or function is explained for MPW C and MPW Pascal; Assembly programmers can use both.

- The calls required to use the performance-measurement tools are now included in Chapter 14 of this book. Examples are shown in MPW C, MPW Pascal, and MPW Assembler.

What you'll need

This section describes the hardware and documentation you need to develop software with the Macintosh Programmer's Workshop 3.0.

Hardware and system requirements

The Macintosh Programmer's Workshop 3.0 can generate applications that run on any Macintosh, including the Macintosh II, Macintosh SE, Macintosh Plus, Macintosh 128K, Macintosh 512K and 512K enhanced, and Macintosh XL.

However, the MPW 3.0 system requires, at the minimum, a Macintosh Plus with 2 megabytes of RAM and a hard disk drive. MPW does not run on the Macintosh XL, the Macintosh 128K, the Macintosh 512K, or Macintosh 512K enhanced or on systems without hard disks. MPW 3.0 requires the 128K or 256K ROMs; it cannot execute on the older 64K ROMs. The ideal developmental system for use with MPW 3.0 is a Macintosh II with an 80-megabyte SCSI hard disk drive, 4 or more megabytes of memory, and System 6.0.2 or later software with MultiFinder.

In general, a small RAM cache of about 32K is useful. Use of MPW with Switcher™ is *not* supported.

MPW software is shipped on 800K disks. Although MPW 3.0 can still read from and write to disks that use the nonhierarchical filing system, MPW's files must be kept on disks that use the hierarchical filing system (HFS). Hard disks, when used as boot disks, must be HFS volumes.

Apple's Macintosh peripherals, including the LaserWriter family of printers and the AppleShare® file server, are supported.

System Folder requirements

Please make sure that you are using System file Version 6.0.2 or later versions.

MPW 3.0 requires these minimum system file versions:

- System file 6.0.2
- Finder 6.1
- Laser Prep 4.0
- ImageWriter® 2.6
- AppleTalk® ImageWriter 3.1
- LaserWriter 4.0

These files are available on version 6.0.2 or later of the *System Tools* disk, and on the latest version of the *Printer Installation* disk.

Documentation

In addition to the *MPW 3.0 Reference*, you should have the *SADE Reference*, the *Macsbug Reference*, and the *ResEdit Reference*. These books together make up the MPW 3.0 documentation suite.

The four MPW programming languages, MPW Assembler, MPW C, MPW C++, and MPW Pascal, are available as separate products.

All programmers need Volumes I–IV of *Inside Macintosh* (published by Addison-Wesley, 1985), the definitive guide to the Macintosh Operating System and user-interface toolbox. Additional features of the Macintosh SE and Macintosh II computers are documented in Volume V. If you need to understand and control the numeric environment, make sure that you have the *Apple Numerics Manual*, a guide to the Standard Apple Numerics Environment (SANE™). Finally, you need the appropriate documentation for the programming language you use:

- **Assembly language:** *Macintosh Programmer's Workshop 3.0 Assembler Reference.* This reference is part of a separate product available from Apple. You may also need the appropriate microprocessor documentation from Motorola.

- **C:** *Macintosh Programmer's Workshop 3.0 C Reference.* This reference is available as part of a separate MPW product. For a guide to the C language itself, you'll need *The C Programming Language* by B. Kernighan and D. Ritchie, or a similar C manual.

- **C++:** *MPW 3.0 C++ Reference.* Also recommended is *The C++ Programming Language* by Bjarne Stroustroup.

- **MacApp:** *MacApp Programmer's Reference.* This reference is part of a separate product, MacApp, the Expandable Macintosh Application, available from Apple. The MacApp product also requires MPW Pascal or MPW C++.

- **MacsBug:** *MacsBug Reference.* This reference is included as part of the MPW 3.0 product.

- **Pascal:** *Macintosh Programmer's Workshop 3.0 Pascal Reference.* This reference is available as part of a separate MPW product.

- **ResEdit:** *ResEdit Reference.* This reference is included as part of the MPW 3.0 product.

- **SADE:** *SADE Reference.* This reference is included as part of the MPW 3.0 product.

About this reference

Part I of this book describes the MPW development system, including the Shell and tools. Part II of this book is a complete alphabetical reference to MPW commands that may be removed to a smaller binder for easy reference.

This reference is written for programmers who are already familiar with the Macintosh. It outlines the process of building a program but does not deal with the particulars of writing it. Language-specific information is covered in the appropriate language references. Language-specific examples in this reference are given in MPW Assembler, MPW Pascal, or MPW C.

If you are new to MPW, be sure to read the Overview in Chapter 1 and the brief section "Building a Program: An Introduction" in Chapter 2. This introduction will take you through MPW's build process in minutes. Chapter 3 introduces the commands available from the menus and Chapter 4 covers the basics of using MPW, including features of the Commando dialog interface.

If you are a seasoned MPW user, this introduction should be sufficient to alert you to the changes to the MPW Shell since MPW 2.0, and to indicate where you can find complete details on each innovation. You may wish to read the new Chapter 7, "Projector: Project Management." Please note that Link and Make are now described in their own chapters in this reference and that ResEdit and MacsBug are now documented in separate volumes. More examples have been added since MPW 2.0, and suggestions from readers have been incorporated to make it easier to find information.

Finding information fast

During MPW sessions, the on-line Help files included with MPW are your first recourse. If you don't find the information you need there, the recommended procedure is to check the Table of Contents and then the index at the end of Part I in this reference. Use the color-keyed tabs to turn quickly to the section in the *MPW Reference* that you need. Then use the table of contents provided at the beginning of each chapter.

The index has been redesigned for MPW 3.0. A single datum in the text (excluding appendixes and Part II) may be referenced from as many as six different points in the Index and up to three levels deep. References include practical task-oriented identification to help you find exactly what you need without looking up a series of page references for a single word. Trivial references have been eliminated from the index to help you avoid wild-goose chases. Examples, tables, warnings, and special notes have been listed to help you find things you may have encountered before but can't remember exactly where.

Throughout this book you will encounter supplementary background information, hints, and tips in specially formatted boxes set off by diamond-shaped icons and sans-serif type. You can ignore these boxes during routine reference.

In spite of redundancy and a plethora of cross references, finding a specific item of information in a book this size can sometimes be frustrating. A little preparation can help out later when you are busy and need to find something fast. It's a good idea to begin by carefully studying the organization of the Contents pages, especially the List of Figures and Tables and the appendixes at the end of Part I. The List of Figures and Tables and the appendixes are often overlooked. You may find it useful to glue tabs at the locations of important figures and tables. Whenever you come across something in the body of the text that you think you may need to find later, place a tab there and label it.

Part II of this manual is a complete alphabetical reference to MPW commands. As you become familiar with MPW and no longer need to refer often to the indexed chapters of Part I, you may find it convenient to remove Part II and place it in a smaller binder for handy reference. You may want to include some of the appendixes (such as the summary of the Resource compiler's syntax in Appendix D) in the smaller binder also.

Syntax notation

The following syntax notation is used to describe MPW commands:

`code`	Courier text is used in examples to indicate characters that must appear in a command line exactly as shown. Special symbols (–, §, &, and so on) must also be entered exactly as shown. Command-line examples are always set off in separate paragraphs.
`include`	Command-language identifiers and syntax elements are set in Courier to differentiate them from surrounding Garamond text (following the Kernighan and Ritchie notation conventions).
nonterminal	Items in italics can be replaced by anything that matches their definition. When referred to in the text, variables normally appear in italics.
{FontSize}	Standard MPW Shell variables appear without spaces between braces.
[optional]	Brackets mean that the enclosed elements are optional.
–o	Hyphenated command-line options appear in boldface when mentioned in text.
repeated...	An ellipsis (...), when it appears *in the text of this reference only*, indicates that the preceding item can be repeated one or more times. Do not confuse this reference convention with the ellipsis command-line character (Option-Semicolon), used to invoke the Commando dialog interface.
a | b	A vertical bar indicates an either/or choice.
(grouping)	Parentheses indicate grouping (useful with the | and ... notation).

This notation is also used in the output of the Help command. (See "The Help Command" in Chapter 4.)

Filenames and command names are not sensitive to case. By convention, they are shown with initial capital letters. Important terms are printed in **boldface** when they are first introduced and defined; these terms are also fully defined in the glossary. Proper names of key user-interface elements, such as the Shell, appear with initial capitals. Command-key or option-key commands (such as Option-L) are always defined in the text with capitals for clarity; nonetheless, the commands work with lower case letters.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning**   Warnings like this indicate potential problems. ▲

△ **Important**   Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents important points that should not be overlooked.

◆ **Hints**

Text set off in this manner in Helvetica type indicates practical hints or background information that need not be perused during routine reference. ◆

For more information

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. (MPW is distributed through APDA.) For information about APDA, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299

1-800-282-APDA, or 1-800-282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink: DEV.CHANNELS

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51-W
Cupertino, CA 95014-6299

Chapter 1 **System Overview**

THIS CHAPTER IS A GUIDE TO THE STRUCTURE OF THE MPW 3.0 SYSTEM and an introduction to its components. If you are new to MPW, this chapter will help you to get oriented. The MPW Shell commands and the MPW tools are grouped according to task each tool or command is briefly introduced and cross-referenced. ∎

Contents

The MPW Shell

The **MPW Shell** is an application that provides an integrated, window-based environment for program editing, file manipulation, compiling, linking, and program execution. The other parts of the Macintosh Programmer's Workshop 3.0—the language and resource compilers, debuggers, Projector, Commando, and other tools described below (except independent applications such as ResEdit)—operate within the Shell environment. These tools accept input from files and Shell windows, and direct output to them.

The Shell combines a command language, a text editor, the Commando user interface, and the Projector project-management system. You can enter commands in any window, even within an ordinary text file, or you can execute them by using menus and dialogs. (A **dialog** may include one or more **dialog boxes,** which may in turn contain text boxes, check boxes, radio buttons, and so on.) For every MPW tool there is a Commando dialog offering all parameters, functions, and options of the command language along with built-in context-sensitive help.

The command language provides text-editing and program-execution functions, including parameters to programs, scripting (command file) capabilities, input/output redirection, and structured commands. You run a tool by typing its name, and then a list of options and affected files. You can link tools together in custom scripts, piping the output of one to the input of another, thereby automating complex operations.

The window operations, menus, and menu items are easily customized to fit your specific needs or preferences.

The MPW Shell integrates the following functional components:

- **An editor** for creating and modifying text files. The editor implements normal Macintosh-style editing together with scriptable editing commands so that you can program the Shell to perform editing functions. (See Chapters 3, 4, and 6.)
- **A command interpreter** interprets and executes the commands you enter in a window or read from a file. (See Chapter 5 and Part II.)
- **The Commando user interface** displays dialog boxes providing immediate, mouse access to all of MPW's many functions, features, and options, including on-line help. (See Chapter 4 for an introduction to the use of the Commando dialogs. Chapter 13 is a guide to creating and editing your own Commando dialogs.)

- **A command interpreter** interprets and executes the commands you enter in a window or read from a file. (See Chapter 5 and Part II.)

- **The Commando user interface** displays dialog boxes providing immediate, mouse access to all of MPW's many functions, features, and options, including on-line help. (See Chapter 4 for an introduction to the use of the Commando dialogs. Chapter 13 is a guide to creating and editing your own Commando dialogs.)

- **Built-in commands,** in addition to editing functions, include commands for managing files without returning to the Finder, commands for manipulating windows, processing variables, command control flow, and more. (See Chapter 5.)

- **Projector, a project-management system,** makes it easy to track the revision history of even large projects with many contributors, with or without a network. Projector helps you avoid confusing versions or getting out of synch with colleagues. (See Chapter 7.)

- **The MPW tools,** over 135 versatile programming tools and scripts designed to run within the MPW environment. Every tool is equipped with a complete dialog interface including context sensitive help. Part II of this reference is an alphabetically organized guide to each of these tools and their many options.

Window commands

All work in MPW is done within windows. The following commands are available for manipulating windows:

Close	Close a window.
MoveWindow	Move window to a specified location on screen.
New	Open a new window.
Open	Open a window.
RotateWindows	Rotate the sequence of a tiled or stacked array of windows.
SizeWindow	Set a window's dimensions.
StackWindows	Arrange open windows in a staggered diagonal array.
Target	Make a window the target window.
TileWindows	Arrange open windows in a tile pattern.
Windows	List windows.
ZoomWindow	Enlarge or reduce a selected window.

File-management commands

The MPW Shell provides the following tools and built-in commands for manipulating files and directories without having to exit to the Finder (see the MPW tool section later in this chapter for other commands that help to manage files):

Backup	Back up folder files.
Catenate	Concatenate files.
Delete	Delete files and directories.
Directory	Set the default directory.
Duplicate	Duplicate files and directories.
Eject	Eject volumes.
Equal	Compare files and directories.
Erase	Initialize volumes.
Exists	Find out if a file or directory exists.
Files	List files and directories.
Mount	Mount volumes.
Move	Move files and directories.
Newer	Compare two files to see which was modified most recently.
NewFolder	Create a directory.
Rename	Rename files and directories.
Save	Save files in edit windows.
SetFile	Set file attributes.
Sort	Sort or merge files.
Unmount	Unmount volumes.
Volumes	List mounted volumes.
Which	Determine which file (pathname) the Shell will execute.

Project-management commands

Projector provides the following built-in commands and scripts for managing projects and tracking revisions. See Chapter 7 for a complete explanation of Projector.

CheckIn	Add or return files to a project.
CheckOut	Check out a file for reading only or for modification.
CheckOutDir	Set location of CheckOut directory.
CompareRevisions	Compare two revisions of a file in a project.
DeleteRevisions	Delete selected revisions and branches of the named files.
DeleteNames	Delete user-defined symbolic names.
MergeBranch	Merge a branch revision onto the trunk
ModifyReadOnly	Change a file checked out as read-only to allow modification.
MountProject	Add the pathname of a project to the root project list.
NameRevisions	Name a set of revisions for the files of a project.
NewProject	Create a new project directory.
OrphanFile	Orphan a file from a project.
Project	Set or write the current project.
ProjectInfo	List current state of all files within a project.
TransferCKID	Transfer resource information in one Projector file to another.
UnmountProject	Remove the pathname of a project from the root project list.

Editing commands

Besides the Macintosh's usual mouse-and-menu editing capabilities, a number of built-in editing commands are provided. You can use these commands both interactively and in scripts. Editing commands feature the use of **regular expressions,** a set of special operators that forms a powerful language for defining text patterns. Other useful commands for editing (such as MatchIt and Translate) are listed later in this chapter under "MPW tools." See "Pattern Matching" in Chapter 6 for a discussion of regular expressions.

Adjust	Adjust lines.
Align	Align text to left margin.
Clear	Delete the selection.
Copy	Copy the selection to the Clipboard.
Cut	Copy the selection to the Clipboard and delete the selection.
Find	Find and select a text pattern.
Format	Specify format of a file (font, tabs, font size).
Mark	Mark and name a text selection.
Markers	List marked selections.
Paste	Replace the selection with contents of the Clipboard.
Position	List the position of selections in a window.
Replace	Replace the selection.
Revert	Revert to saved file.
Undo	Undo last edit.
Unmark	Remove a marker from its text selection.

Structured commands

The Shell also provides a number of built-in structured commands. Used mainly in scripts, these commands provide conditional execution and looping capabilities:

Begin...End	Group commands.
Break	Break from For or Loop.
Continue	Continue with next iteration of For or Loop.
Exit	Exit from a script.
For...	Repeat commands once per parameter.
If...	Conditional command execution.
Loop...End	Repeat commands until Break.

Other built-in commands

The MPW Shell also provides a number of other predefined commands:

AddMenu	Add menu item.
Alert	Display alert box.
Alias	Define alternate command names.
Beep	Generate tones.
Confirm	Display confirmation dialog box.
Date	Write the date and time.
DeleteMenu	Delete a user-defined menu or item.
Echo	Echo parameters.
Evaluate	Evaluate an expression.
Execute	Execute a script without affecting variable scope.
Export	Make variables available to programs and scripts.
Flush	Clear the command cache.
Help	Display summary information.
Parameters	Identify parameters.
Quit	Quit MPW.
Quote	Echo parameters, quoting if needed.
Request	Request text from a dialog box.
Set	Define and write Shell variables.
Shift	Renumber script positional parameters.
ShutDown	Shut down or reboot machine.
Unalias	Remove aliases.
Unexport	Remove variable definition from export list.
Unset	Remove Shell variables.

MPW scripts

The menu commands available in the Directory and Build menus use some of these scripts:

BuildCommands	Show build commands.
BuildMenu	Create the Build menu.
BuildProgram	Build the specified program.
CCvt	Convert pre-3.0 C source to 3.0-compatible source.
CompareFiles	Compare two files side by side, pinpointing any differences.
CompareRevisions	Identify and compare project revisions.
CPlus	Compile C++ programs.
CreateMake	Create a simple makefile.
DirectoryMenu	Create the Directory menu.
DoIt	Highlight and execute a series of commands.
Line	Find specified line in file.
MergeBranch	Merge a branch file back into the trunk of a project.
OrphanFile	Orphan a file from a project.
SetDirectory	Set current directory (from Directory menu).
TransferCKID	Transfer resource information in one Projector file to another.

MPW tools

MPW tools are programs that run within the Shell environment. With the exception of the language compilers, the tools listed here are included with the *Macintosh Programmer's Workshop 3.0;* several are described in more detail in the sections that follow.

Asm	MC68000-family Macro Assembler (available as a separate product).
Backup	Back up folder files.
C	C compiler (available as a separate product).
Canon	Canonical spelling tool.
CFront	Translator for C++.
Choose	Choose or list volumes or printers (scriptable chooser).
Compare	Compare text files.
Count	Count lines and characters.
DeRez	Resource decompiler.
DumpCode	Dump code resources.
DumpFile	Display contents of an arbitrary file as hex and ASCII.
DumpObj	Dump object files.
Entab	Convert runs of spaces to tabs.
FileDiv	Divide a file into several smaller files.
GetErrorText	Display text for system error numbers.
GetFileName	Display a standard file dialog box.
GetListItem	Present file selection list in dialog box.
Lib	Combine object files into a library file.
Link	Link an application, tool, or resource.
Make	Program maintenance tool.
MakeErrorFile	Create error message textfile.
MatchIt	Match paired language delimiters.
Pascal	Pascal compiler (available as a separate product).
PasMat	Pascal program formatter (part of MPW Pascal).
PasRef	Pascal cross-referencer (part of MPW Pascal).
PerformReport	Generate a report analyzing program performance.
Print	Print text files.
ProcNames	Display Pascal procedure and functions names (part of MPW Pascal).
ResEqual	Compare files on a resource-by-resource basis.
Rez	Resource compiler.
RezDet	Detect inconsistencies in resources.

Search	Search files for a pattern.
SetPrivilege	Set access privileges to folders on file server.
SetVersion	Maintain version and revision numbers.
Sort	Sort files.
Translate	Convert one or more characters.
WhereIs	Locate files buried deep in a directory tree.

MPW Assembler

The Assembler is provided as a separate product, MPW 3.0 Assembler, which includes the following:

■ Translation of MC68000, MC68010, MC68020, and MC68030 assembly-language programs into object code

■ Support for MC68881 and MC68882 floating-point instructions and MC68851 memory management instructions

■ Powerful macro facilities, code and data modules, and entry points, local labels, and (optional) optimized instruction selection

■ Assembly-language interfaces to *Inside Macintosh* routines

■ Sample programs

MPW Pascal tools

The Pascal system is provided as a separate product, MPW 3.0 Pascal, which includes the following:

■ Pascal compiler

■ Pascal cross-reference program (PasRef)

■ Pascal source file format program (PasMat)

■ Pascal procedure and name program (ProcNames)

■ Pascal runtime library

■ Pascal interfaces to the *Inside Macintosh* routines

■ Sample programs

Macintosh Programmer's Workshop 3.0 Pascal is an improved version of MPW 2.0 Pascal. The Pascal tools PasRef, PasMat, ProcNames, and the Pascal compiler are also documented in Part II of this reference.

MPW C compiler and C++ translator

The C compiler and C++ translator are provided as separate products. MPW 3.0 C includes the following:

- C compiler
- Standard C Library
- C interfaces to the *Inside Macintosh* libraries
- Sample programs in MPW C

The C Compiler implements the full C language as defined in *The C Programming Language,* by Brian Kernighan and Dennis Ritchie. The usual extensions to this definition provide enumerated types and structure assignment, parameters, and function results. In addition, Apple extensions provide SANE numerics and interfaces to Pascal functions and Macintosh traps. The compiler supports many ANSI C features, such as function prototypes and strict pointer compatibility. Most Standard C Library functions, including character and string processing, memory allocation, and formatted input/output, are also provided.

MPW 3.0 C++ includes the following:

- C++ translator (CFront)
- C++ Streams Library
- Sample programs in MPW C++

The CFront translator from AT&T implements the full C++ language as defined in *The C++ Programming Language,* by Bjarne Stroustroup. The current version, CFront 2.0, also implements multiple inheritance and other extensions described in the paper "Evolution of C++ from 1985 to 1987" by Bjarne Stroustroup. In addition to the C extensions listed in that paper, C++ also contains extensions that allow C++ to be used with MacApp.

Link

The linker (Link) combines object code files into executable programs, driver resources, or stand-alone code resources. Link includes, by default, only the code and data modules that are referenced. Link replaces the code segments in an existing resource file, without disturbing other resources in the file. An option directs Link to produce a link map as a text file. Other options allow the creation of an object module cross-reference file, a file containing a list of all the unreferenced modules, and a symbolic debugger file.

A separate tool, Lib, provides library manipulation. Linking is performed automatically for simple programs constructed by using the Build menu. Chapter 8 describes the use of Link in building a program. See Chapter 10 for more details on the operation of the linker.

Make

The Make tool simplifies software construction and maintenance. Its input is a list of dependencies between files and instructions for building each of the files. Make generates commands to build specified target files, rebuilding only those components that are out-of-date with respect to their dependencies. You can generate makefiles automatically from commands in the Build menu. To use Make with more elaborate programs, see Chapter 9.

Throughout this reference examples demonstrating Make or makefiles assume that you are using Apple's MPW languages. Because Make assumes certain default rules that apply only to Apple's MPW languages, you may need to make modifications for non-Apple programming languages. Please consult your compiler's documentation for instructions on how to modify these default rules.

Resource compiler and decompiler

The resource compiler (Rez) reads a textual description of a resource and converts it into a standard Macintosh resource file. The resource decompiler (DeRez) converts resources into a textual representation that can be edited in the Shell, and recompiled with Rez. You can use DeRez to create resource compiler input from any existing resource files. Rez and DeRez need templates (type declarations) to define resource types. Definitions of the standard Macintosh resource types ('MENU', 'STR#', 'ICON', and so on) are provided in two commented text files, Types.r and SysTypes.r. Another tool, RezDet, checks resource files for consistency (see Part II). Rez and DeRez are documented in Chapter 11.

Rez's capabilities have been extended in MPW 3.0. Two new functions let you delete resources or change resource types from within Rez. The new syntax element Label has been supplied to support more complex resources, such as those found in color QuickDraw.

Commando

The Commando tool implements the Commando dialog user interface for all MPW tools and commands. Obviously, this is a great convenience for dealing with tools offering many interdependent options. Newcomers to MPW will appreciate Commando's instant assistance in building complex command lines. The dialogs include a Help frame with information on each selected data field or control. You can also use Commando to create specialized dialogs for your own MPW tools and scripts.

Commando looks in a tool's or script's resource fork for a resource of the type `'cmdo'`. Commando then loads the resource, builds a dialog, handles events, and passes the resulting command line back to the Shell for execution. The basics of using Commando dialogs are described in Chapter 4. Dialogs utilizing specialized types of dialog boxes are presented with the tools they support in Part II. Chapter 13 tells you how to create a Commando interface for your own tools and scripts.

Projector

Projector is an easy-to-use project-management system that can be customized to fit any working style, from the single programmer to the large networked engineering team. Use Projector's file-locking feature to control changes to master files, track a project's revision history, and generally keep your projects organized.

Briefly, here's how it works: You begin a work period by checking out a file from Projector for either review or modification. Although many people can review a file, only one person at a time can modify a file. When you've finished your work, you check any modified files back in with Projector, along with a note detailing your modifications. Your name, your notes, and the date are automatically filed in Projector's revision history for that project. Various branches of a file containing different modifications may be later merged into one master file.

Projector's commands (listed in the section "Project-Management Commands" earlier in this chapter) are built into the Shell. Chapter 7 is a detailed account of Projector.

Conversion tools

Canon is a tool for regularizing the spelling and capitalization of identifiers in source files moved from other systems. (In MPW languages, all characters are significant rather than just the first eight as in the Lisa Workshop. In C, case is also important.)

The file Canon.dict contains the correct spelling and capitalization for *Inside Macintosh* ROM routines. C programmers, in particular, will find Canon and Canon.dict useful.

Entab is a useful tool for converting space characters and tabs to conform to MPW editor or other editor conventions.

You can look up these conversion tools in Part II.

Performance-measurement tools

The performance-measurement tools enable you to pinpoint where your code is spending time. These libraries allow you to sample the program counter, produce a file of output data, and analyze that data with a report generator. Advanced programmers will find these tools useful for streamlining the execution of their code. Chapter 14 is devoted to this subject. Examples of the actual calls and procedures are presented in MPW C and MPW Pascal.

Applications

Applications are stand-alone programs that can execute outside the Shell environment. SADE and ResEdit are both stand-alone programs provided with MPW. It is assumed that you already have the Font/DA Mover, which is distributed on the system tools and system installation disks. Any application can be executed from the MPW Shell.

ResEdit

ResEdit is an interactive, graphically based editor for creating, editing, and copying resources. An interface like that in the MacDraw application is provided to help you design your own fonts. ResEdit includes a set of routines that make it possible to write your own add-on resource editors for ResEdit. See the separate *ResEdit Reference* for a thorough explanation of ResEdit.

SADE and MacsBug

The new Symbolic Application Debugging Environment (SADE) is a symbolic debugger with an interactive graphic interface like that of the MPW Shell. SADE is an application that runs under MultiFinder and can be used to debug other applications and MPW tools. You can monitor the execution of your program simultaneously at the processor level and the symbolic program source level. This first release of SADE includes

- source display
- variable display according to type
- display of Macintosh system structure
- source level breaks and stepping
- programmable, extensible command language

SADE is included with MPW 3.0 but documented separately in the *SADE Reference*. See Appendix F of this reference for the object file format.

The familiar MacsBug has been improved for MPW 3.0, and is also documented in a separate volume, *MacsBug Reference*.

MacsBug fully supports the MC68000, MC68020, and MC68030 processors, as well as the MC68881, MC68881, and MC68851 coprocessors. MacsBug resides in RAM together with your program. MacsBug allows you to examine memory, trace through a program, or set up break conditions and execute a program until they occur. MacsBug runs on all Macintosh computers with 128K or larger ROMs, including the Macintosh SE and Macintosh II. See the *MacsBug Reference* for instructions on using MacsBug.

Special scripts

Several special command scripts are provided. They are essential for operation of the MPW Shell. These text files contain commands that are read by the Shell at startup and shutdown:

- The Startup file is a command script that calls another script, UserStartup, that is run each time you start the MPW Shell. You can use UserStartup to customize MPW. The Startup file now executes UserStartup and then any file named UserStartup•*name* in the directory that contains the Shell. (Press Option-8 to obtain the • symbol.) If you have a customized UserStartup file, you may want to personalize it (for example, UserStartup•Tom) so that when you install MPW 3.0 your customized file won't be overwritten. The Startup file is discussed in detail in Chapter 5.

- The Suspend and Resume files are scripts that preserve the state of the Shell environment while a stand-alone application is executing. The Quit file saves the state of the Shell environment when you exit to the Finder.

Examples

In addition to the examples excerpted in this reference work, you'll find numerous complete examples in the Examples folder included on the MPW distribution disks. The examples are written in MPW C, MPW Pascal, and MPW Assembler. Examples illustrating the use of Projector are also included in this folder. If you are using a different compiler sold with MPW 3.0, check the compiler's documentation and distribution disks for specific versions of these sample programs. See Appendix A for the location of the MPW 3.0 Examples folder.

Sample program files

Source files are provided for sample MPW tools and desk accessories. Versions of these sample programs are included in MPW Assembler, MPW C, and MPW Pascal. They can be found in the Examples folder. The Examples folder also contains instruction files and makefiles for building the sample programs. Some of these examples are referred to in Chapter 2, "Building A Program: An Introduction."

Note that these sample files are part of the respective MPW C, MPW Pascal, and MPW Assembler products.

Command-language examples

Examples of the use of the MPW command language are provided in the folder Examples. Among these are

■ addmenu commands for creating user-defined menu items

■ a list of UNIX-oriented aliases

■ suggestions for modifying the Startup script

To learn more about these examples, open the file Instructions in the Examples folder. Additional examples are included with each of the MPW commands in Part II of this reference. The command language is documented in Chapter 5.

Overview of MPW files and directories

Appendix A contains a complete list of all of the Macintosh Workshop 3.0 files. It also describes the recommended setup of files on a hard disk. Figure 1-1 shows the MPW folder layout. Folders for the Pascal, C, and Assembler systems are also shown, along with folders for your applications and projects.

■ **Figure 1-1**   Setup of MPW folders and files
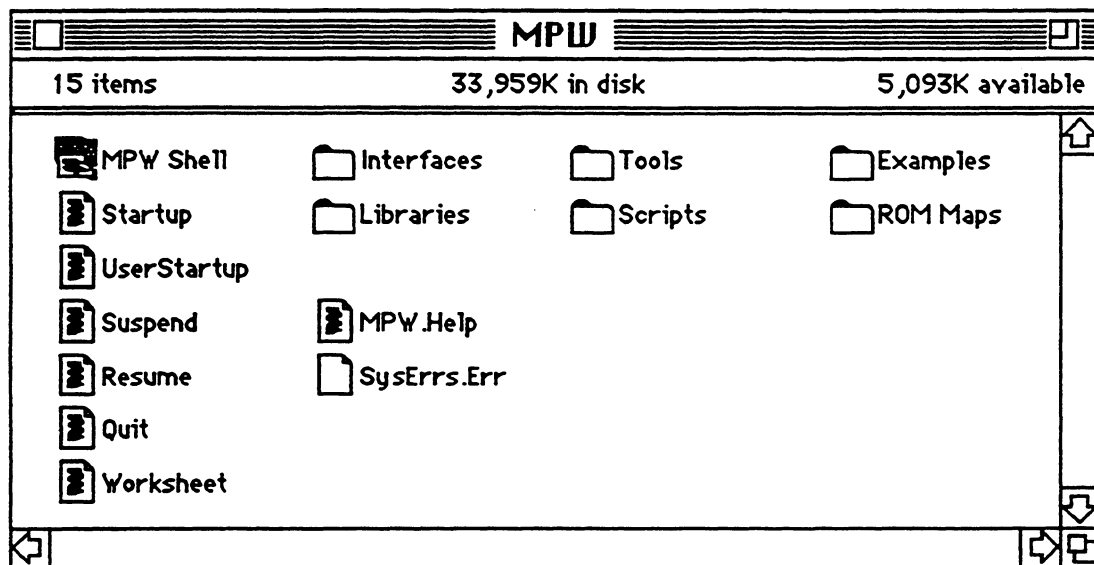
Be sure to see "Installing the System" in Chapter 2.

Chapter 2 **Getting Started**

THIS CHAPTER EXPLAINS HOW TO START USING MACINTOSH PROGRAMMER'S WORKSHOP
3.0. Even if you are familiar with MPW 2.0, it's a good idea to read the next
section that describes the new automated installation procedure. (You might run
into some pathname conflicts if you simply drag files from the 3.5-inch disk to
your hard disk.) This chapter also contains the section "Using MPW With
MultiFinder," which explains how to use MPW while running a compiler in the
background. You'll also find a section with guidelines for sharing MPW from a file
server.

Basic rules of operation are introduced here and in Chapters 3 and 4. If you are
new to MPW, the tutorial "Building a Program: An Introduction," later in this
chapter, will introduce you to the simplicity of using this environment. ▪

Contents

Installing the system

Macintosh Programmer's Workshop 3.0 is shipped on five 800K disks: *MPW1, MPW2, MPW3, MPW4,* and the *MPW Installation Disk. (MPW Assembler, MPW Pascal, MPW C,* and *MPW C++* are separate products.)

Before attempting to install MPW, please check the section "Hardware and System Requirements" in the Introduction of this book.

Appendix A, "Macintosh Programmer's Workshop Files," contains an annotated list of MPW files and shows the recommended arrangement of files on a hard disk. Pathname rules for the Hierarchical File System (HFS) are explained later in this chapter. Also see Figure 1-1 at the end of Chapter 1 for a suggested arrangement of MPW folders and files.

A complete MPW 3.0 system, including all three MPW languages, requires over 6 megabytes of disk space.

MPW 3.0 includes an Installer script on the MPW Installation Disk, for systematically installing the complete MPW system from the other four disks so that everything is located in the folders that MPW expects. You need at least 6 megabytes of space on your HFS hard disk to complete the full installation. However, the Installer does give you the option of stopping the installation before all of the tools on disks *MPW3* and *MPW4* have been installed.

▲ **Warning**   Don't simply drag the MPW Shell or any other files from the Installer disk to your hard disk. The files on the Installer disk are used for automatic installation only, and thereafter you'll discard them. ▲

To automatically install MPW 3.0, follow these steps:

1. Insert the *MPW Installer* disk in the 3.5-inch disk drive.

2. Drag the Installation folder to your hard disk. If you have multiple hard disks, drag the folder to the hard disk on which you want MPW to reside.

3. Open the folder and double-click the icon labeled "MPW Installer."

4. The first Installer dialog box appears:

```
┌─────────────────────────────────────────────────────────────┐
│ ┌─────────────────────────────────────────────────────────┐ │
│ │ This is the installation procedure for MPW 3.0.         │ │
│ │ "Internal:MPW:" will be installed. Insert the first     │ │
│ │ MPW distribution disk in drive 1 and click OK.          │ │
│ │ ┌──────────────┐              ┌──────────────┐          │ │
│ │ │      OK      │              │    Cancel    │          │ │
│ │ └──────────────┘              └──────────────┘          │ │
│ └─────────────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────────────┘
```

5. Click OK and insert the distribution disks in any order. The Installer program creates a folder named MPW at the root directory of the volume in which the Installer folder is located.

6. When the installation is complete, or when you have clicked a Cancel button, the Installer quits the Shell. Now throw away the Installation folder. You are left with MPW in a folder at the root directory, ready to go.

The order in which the disks are copied doesn't matter, and it's okay to insert the same disk more than once. You may also choose to stop by clicking the No button before you've copied all the distribution disks.

If you decide to click the Cancel button for any reason, the MPW Shell Worksheet appears. (In that case, after quitting MPW, don't save the Worksheet file that was created during the installation. It's better to start all over again.)

▲ **Warning**   Don't use apostrophes or any other special characters in the hard disk volume name. This would cause the Installer to fail. ▲

Using MPW with MultiFinder

It would be very convenient to be able to work in the Shell or editor while waiting for a compiler to run in the background. But MultiFinder lets you switch to different applications only while running a tool; you cannot normally work in the Shell or editor while running a tool in the background.

However, you can obtain this virtual multitasking capability by configuring a second MPW Shell. You work in the second Shell while the first maintains the background operation of any tool or script. Here is a way to set up the second MPW Shell:

Create a folder called Concurrent MPW and put these files in it:

- **MPW Shell**

  Be sure to rename the second MPW Shell in this directory to something like "Concurrent Shell" or perhaps "MPW Editing Shell" so that you can quickly identify which Shell you are currently using.

- **Startup**

- **UserStartup**

  This file isn't crucial, but without the variables, aliases, and menus defined in your UserStartup, the Concurrent Shell would not be configured to your normal working environment.

- **MPW.Help**

  Alternatively, you could keep just one copy of MPW.Help in your main MPW directory and use an alias in your Edit MPW. For example: alias help 'help -f HD:MPW:MPW.help'.

- **SysErrs.Err**

  If you get an error from MPW and don't have a copy of this file, you'll see an error message such as:

  ```
  # OS error -43 (Error message file not available)
  ```

- **Quit**

You can now use this second MPW Shell system while tools are running concurrently in the first MPW Shell. This configuration is only a suggestion. You could simplify it a bit, as indicated in the preceding notes. Also, the memory size in the second Shell may be decreased to 512K if it is used only for editing and small tools.

♦ *Note:* Although you cannot move Shell windows or pull down menus while a tool is running, remember that you can switch applications by clicking the application icon in the menu bar.

The same file cannot be opened for editing by both Shells at the same time.

It's a good idea to generate a sound (using Beep or other tools) at the end of scripts so that you know when your background operations are completed.

Using MPW on a file server

To set up MPW in a shared environment, install the MPW system on the file server. The following files must reside on each workstation that shares the MPW system.

- **MPW Shell**

- **Startup**

- **UserStartup**
  Alternatively, you can change Startup to execute a UserStartup on the file server.

- **MPW.Help**
  Alternatively, you can keep just one copy of MPW.Help on the file server by setting an alias in your Startup file. For example:

  ```
  alias help 'help -f  SharedServer:MPW:MPW.Help'
  ```

- **SysErrs.Err**
  If you get an error from MPW and don't have a copy of this file, you'll see an error message such as:

  ```
  # OS error -43 (Error message file not available)
  ```

- **Suspend/Resume**
  You need these files only if you are not running MultiFinder.

- **Quit**

Starting up

Start up MPW just as you would start any standard Macintosh application.

- ◆ *Note:* A small RAM cache (perhaps 32K) is useful when running MPW 3.0. You may use larger caches if you have plenty of memory. However, some functions in MPW 3.0 may run more slowly with large RAM caches. Use of MPW with Switcher is not recommended; use MultiFinder.

From the Finder, select and open the MPW Shell icon. The **Worksheet window** (shown in Figure 2-1) will appear with its full pathname in the title bar (for example, "HD:MPW:Worksheet"). This window has no close box and is always present on the screen; otherwise it's just like any other window. The Worksheet is your home base. You'll use it most often to type commands and see the return output. You can also write and compile sections of code or keep a diary—anything in the Worksheet can be saved to any window or file.

You can also start MPW by double-clicking any MPW document or tool.

■ **Figure 2-1**   Worksheet window
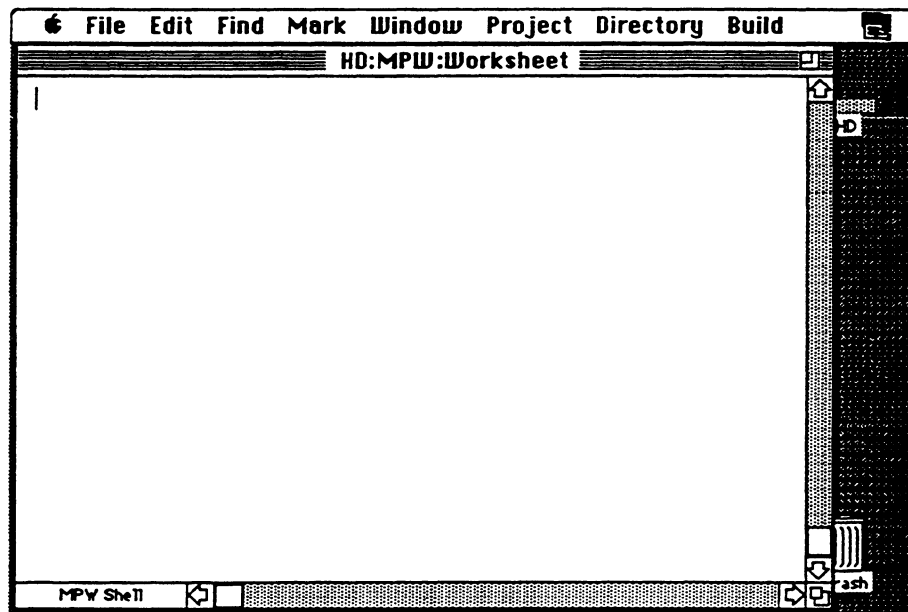
The menus available from the Shell appear in the menu bar at the top of the screen. An explanation of each menu is provided in Chapter 3. You can easily add your own menu names. (See Chapter 8.)

A **status panel** at the window's lower-left corner shows the name of the command that's currently executing, or simply "MPW Shell" when you're not executing a command. A mouse click on the status panel is equivalent to pressing the Enter key.

When you first start the Macintosh Programmer's Workshop, a script called Startup executes. The Startup file defines several variables and command aliases (alternative command names); this file is further described in Chapter 5.

△ **Important**   The Startup file must be in the same directory as the MPW Shell. See Figure 1-1, "Setup of MPW folders and files," at the end of Chapter 1 for an illustration of how your root MPW folder should appear. △

Selecting commands from menus

In MPW, commands may be built-in commands, scripts, tools, or applications, as explained in Chapter 1.

Several of the built-in commands can be executed by using the File, Edit, Mark, and Window menus. The Project, Directory, and Build menus are optional, and are normally installed by UserStartup scripts. Some items in these menus execute scripts (see Chapter 3 for details about menus). These scripts must be located in a folder with a path in the {Commands} variable.

You can add your own menu items to the File, Edit, Find, Directory, and Build menus. By using the AddMenu command you can even add your own menus. Each user-defined menu item specifies a list of MPW commands that are executed when the menu item is selected. See the file AddMenu in the Examples folder for a number of ideas for user-defined menus.

■ **Figure 2-2**   MPW menu bar with MultiFinder


```
   File   Edit   Find   Mark   Window   Project   Directory   Build        ▣
```
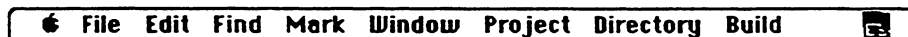
Building a program: an introduction

This section takes you step by step through the process of building a sample program. You'll find that the Build menu and the Commando dialog boxes make the learning process intuitive and comfortable. Even if you've never used MPW before, you can immediately use the Build menus to build programs.

MPW's automated Build menu lets you assemble, compile, and link simple programs without studying the command language, the numerous compiler and Linker options, or countless other details. You can use the Build menu to build applications, stand-alone code resources, desk accessories, and tools written in MPW Assembly language, MPW C, MPW C++, MPW Pascal, and Rez, or in a combination of these languages. You can include resource specifications when building programs with these menus.

The sample programs

In this introduction, three assembly-language programs included with MPW Assembler are suggested as examples:

- **Sample:** the "Inside Macintosh" sample application
- **Count:** an MPW tool that counts characters and lines in a file (see Part II)
- **Memory:** a sample desk accessory that displays the memory available in the application and system heaps and on the boot disk

Similar program examples are included with MPW C and MPW Pascal. If you are primarily interested in programming in one of these languages, be sure to read, in the corresponding language reference, the section on the example programs. If you are using a different (non-Apple) compiler, be sure to check its documentation for information on specific language versions of these examples.

You can routinely rebuild more complex programs by selecting a single menu item. There is a smooth transition from the simple builds to the more complex ones. (See Chapter 8 for information on how to modify the Build menu and the makefile that it creates.)

The source files for each of these three assembly-language examples are in the Examples:AExamples folder that is included with the MPW Assembler distribution disks. For example, the source for Count consists of the files Count.a and FStubs.a. A makefile that contains the commands for building all of the examples is also provided in the same folder. Instruction files are also provided on the MPW disks for each language. If you are new to MPW, we recommend that you start with the tutorial that follows rather than with the Intructions file on the disks. At the conclusion of this tutorial you will be referred back to the disk instructions.

Two easy steps

You can build each of the example programs in two steps, using the Directory and Build menus:

1. Set the current directory.

2. Build the program.

Both of these steps are explained next. You can use this section to take MPW on a test drive.

1. Set the current directory.

Open the Directory menu. The upper half of the menu contains the commands to show the current directory and to change it to an arbitrary directory. (See Figure 2-3.) The lower half of the menu lists frequently used directories.

■ **Figure 2-3**   Directory menu


```
          Directory
┌─────────────────────────────────┐
│ Show Directory                  │
│ Set Directory...                │
├─────────────────────────────────┤
│ HD2:MPW:Examples:AExamples:     │
│ HD2:MPW:Examples:CExamples:     │
│ HD2:MPW:Examples:CPlusExamples: │
│ HD2:MPW:Examples:Examples:      │
│ HD2:MPW:Examples:PExamples:     │
│ HD2:MPW:Examples:Projector Examples: │
│ HD2:MPW:                        │
└─────────────────────────────────┘
```
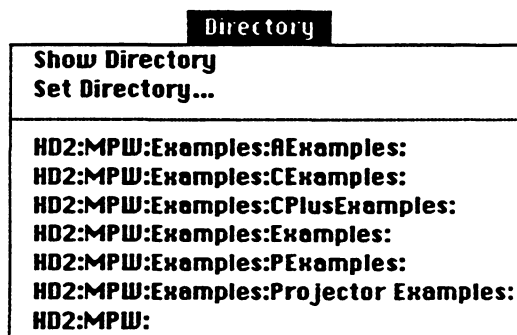
Select Show Directory to find out what your current directory is. You'll see the alert shown in Figure 2-4.

■ **Figure 2-4**   Show Directory alert

```
┌──────────────────────────────────────────────┐
│  ┌────────────────────────────────────────┐  │
│  │                                        │  │
│  │  The default directory is              │  │
│  │                                        │  │
│  │  HD:MPW:                               │  │
│  │                                        │  │
│  │                                        │  │
│  │                          ╭──────────╮  │  │
│  │                          │    OK    │  │  │
│  │                          ╰──────────╯  │  │
│  └────────────────────────────────────────┘  │
└──────────────────────────────────────────────┘
```
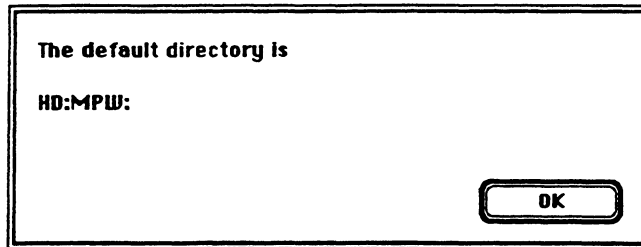
Click OK to remove the alert. You're going to build the assembly-language program Sample, so you'll need to set the current directory to the directory that contains the assembly-language examples. Now open the Directory menu again and select "AExamples." Selecting "AExamples" from the Directory menu runs commands that set the current directory. You can check to see if the current directory has been correctly reset by selecting the Show Directory menu item again. (The Set Directory... menu item is used to add other directories to the list at the bottom of the Directory menu. This menu item is explained in "Building a New Program" later in this chapter.)

2. Build the program.

Now open the Build menu, shown in Figure 2-5, and select any one of the four Build menu items.

■ **Figure 2-5**   Build menu

```
      ┌──────────┐
      │  Build   │
┌─────┴──────────┴───────────┐
│  Create Build Commands...  │
├────────────────────────────┤
│  Build...              ⌘B  │
│  Full Build...             │
│  Show Build Commands...    │
│  Show Full Build Commands...│
└────────────────────────────┘
```
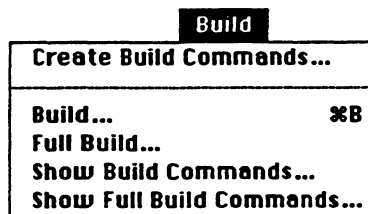
Each Build item builds your specified program in a slightly different way:

Build
The program is built automatically, but only files that have been modified since you last built the program will be processed. Use this item to save time. The Command-key equivalent is Command-B.

Full Build
The program is completely built, ignoring any object files or intermediate files that may exist from a previous build.

Show Build Commands
The commands needed to build the program (using just those files affected by modifications since the last build) are displayed on the worksheet, but not executed. You can then select any or all of the commands—or modify them—and then press Enter to execute them.

Show Full Build Commands
All the commands needed to completely rebuild the program (whether modified since the last build or not) are displayed on the worksheet, but not executed. This is a convenient way to see all of the commands used in building the program you've selected.

See "Build Menu" in Chapter 3 for more information on Build menu items. When selected, each Build item first displays a dialog box like that in Figure 2-6, requesting the name of your program.

For this tutorial, select Full Build.

■ **Figure 2-6**   Program Name dialog box

```
┌──────────────────────────────────────────┐
│ Program Name?                             │
│ ┌──────────────────────────────────────┐ │
│ │I                                     │ │
│ └──────────────────────────────────────┘ │
│  ╭──────────╮          ╭──────────╮       │
│  │   OK     │          │  Cancel  │       │
│  ╰──────────╯          ╰──────────╯       │
└──────────────────────────────────────────┘
```
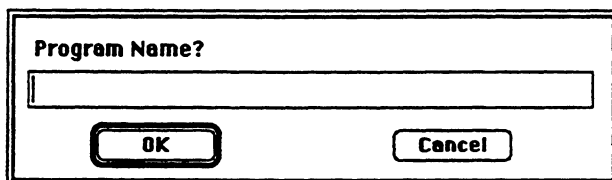
When the Program Name dialog box appears, type the name of the program you want to build (in this case, type "Sample") and then click the OK button. (Be sure that you type the name Sample and *not* Sample.a. Since you have already set the directory to AExamples, you don't need to indicate that you want to build the assembly-language version of Sample. If you give Sample.a as the program name, the Build script will attempt to build the source file.)

The Worksheet window now becomes the frontmost window. The status panel in the lower-left corner flashes the name of each operation as it is performed by MPW. Each of the MPW commands used by the Full Build script appears on the worksheet as it is executed. When the build has finished, your worksheet should look like Figure 2-7.

■ **Figure 2-7**   Finished Sample build

```
 ┌──────────────────────────────────────────────────────────────┐
 │ ¢ File  Edit  Find  Window  Mark  Directory  Build             │
 ├──────────────────────────────────────────────────────────────┤
 │ ════════════════════ HD :MPW:Worksheet ════════════════════   │
 │ ┌──────────────────────────────────────────────────────── ⇧ │ │
 │ │                                                          ▢ │ │
 │ │ ■ 2:21:04 PM ──── Build of Sample.                        │ │
 │ │ ■ 2:21:04 PM ──── Analyzing dependencies.                 │ │
 │ │ ■ 2:21:11 PM ──── Executing build commands.               │ │
 │ │    Rez Sample.r -o Sample -a                              │ │
 │ │    Asm Sample.a                                           │ │
 │ │    Link Sample.a.o -o Sample                              │ │
 │ │ ■ 2:22:08 PM ──── Done.                                   │ │
 │ │    Sample                                                 │ │
 │ │                                                          ⇩ │ │
 │ ├──────────────────────────────────────────────────────────┤ │
 │ │ MPY Shell │⇦│ │                                          ⇨▢ │ │
 │ └──────────────────────────────────────────────────────────┘ │
 └──────────────────────────────────────────────────────────────┘
```
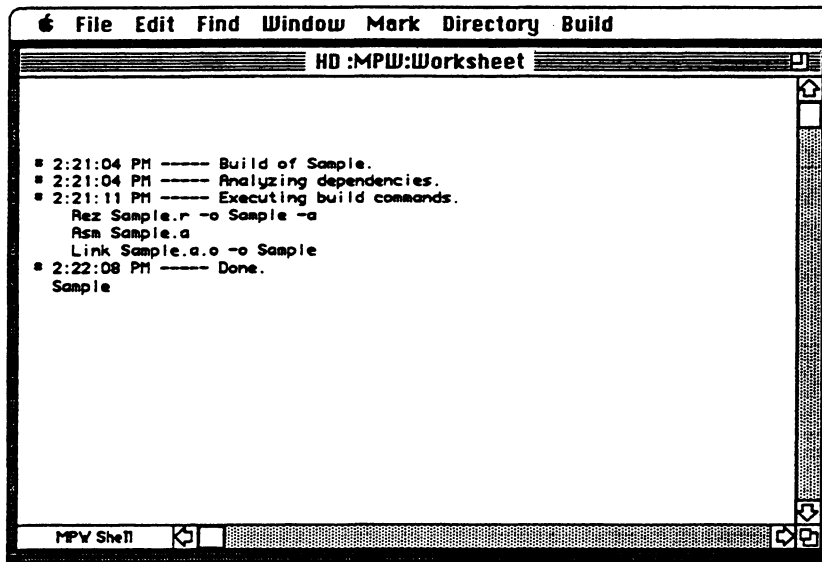
To check your work, press Enter. The Shell then executes the newly built program, displaying the text-edit window that Sample creates (described at the beginning of *Inside Macintosh*). When you quit the Sample program, you are returned to the Shell.

Use the same procedure to build the two other examples in the Examples:AExamples folder: the tool Count and the desk accessory Memory. For guidance in using these examples, consult the file Instructions in the folder AExamples.

In general, to run a newly built program, select its name (and, in the case of a tool, any parameters) and press Enter. If the program you have built is an application, your open windows, user-defined menus, and other status information will be saved before the program is run. When you quit the application you are returned directly to MPW with your previously open windows and menus still displayed. If the program is an MPW tool, it is run without leaving MPW (be sure to specify any required parameters and options).

◆ *Note:* When MultiFinder is running, the application is simply launched in another partition, and the MPW Shell does not exit or go through the Suspend/Resume process.

When you build a desk accessory by using Build or Full Build, the last line of the Build transcript is a command that will run the Font/DA Mover to install the desk accessory in the System file. (Make sure there is enough memory to launch Font/DA Mover.) After this installation is complete, the desk accessory will appear in the Apple menu. If your Font/DA Mover isn't in the directory specified by the {Commands} Shell variable, then you should use either the Finder, the MPW Duplicate command, or the MPW Move command, to move it there.

If you're curious about the functioning of any of the Build commands, see Chapter 8 for more background on the Build process.

Building a new program

The Directory and Build menus are convenient to use when working with your existing programs. You use slightly different steps for creating new programs:

1. Set the current directory by using the Directory menu.

2. Type your program.

3. Select Create Build Commands from the Build menu.

4. Select a build item from the Build menu.

Each of these steps is explained next.

1. Set the directory.

The first step in creating a new program is to set the directory where you want your new program to reside. You can select one of the directories that appears in the Directory menu, or you can select another directory by using the Set Directory menu item. When you select Set Directory from the Directory menu, a standard file dialog box, like that in Figure 2-8, appears.

■ **Figure 2-8**   Set Directory... standard file dialog box


```
┌─────────────────────────────────────────────┐
│  ┌─────────────────────────┐                 │
│  │ Select Current Directory: │               │
│  └─────────────────────────┘                 │
│         ┌──────────┐                          │
│         │ ⊜ MPW   │                          │
│         └──────────┘                          │
│  ┌──────────────────────────┐   ⊂⊃ HD2       │
│  │▣ Examples             ⇧ │                  │
│  │▢ Interfaces             │   ┌──────────┐   │
│  │▢ Libraries              │   │  Eject   │   │
│  │▢ MPW Demo               │   └──────────┘   │
│  │▢ ROM Maps               │   ┌──────────┐   │
│  │▢ Scripts                │   │  Drive   │   │
│  │▢ Tools                  │   └──────────┘   │
│  │                         │   ──────────     │
│  │                       ⇩ │   ┌──────────┐   │
│  └──────────────────────────┘   │  Open    │   │
│                                 └──────────┘   │
│                                 ┌──────────┐   │
│                                 │Directory │   │
│                                 └──────────┘   │
│                                 ┌──────────┐   │
│                                 │ Cancel   │   │
│                                 └──────────┘   │
└─────────────────────────────────────────────┘
```
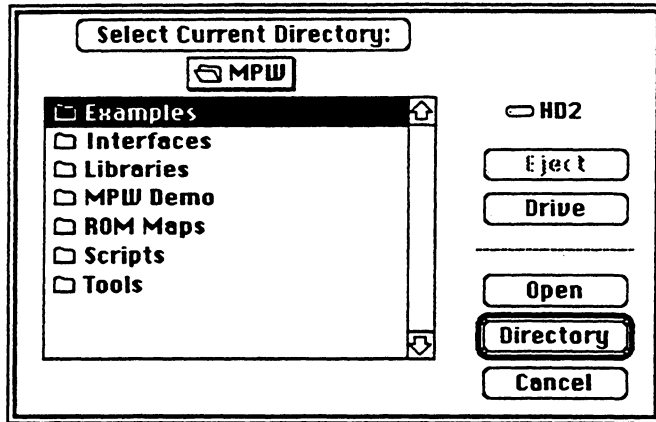
Select the directory you need. After highlighting the directory you want, click Directory or Select Current Directory: at the top of the dialog box. The new directory will then be added to the list of directories on the Directory menu.

2. Type your program.

The next step is to create the source files for your program. Select New in the File menu. (Remember that assembly-language source filenames should end with ".a", C filenames with ".c", C++ filenames with ".cp", Pascal filenames with ".p", and Rez filenames with ".r".) An empty window now appears and you are ready to type your program. Enjoy!

3. Select Create Build Commands from the Build menu.

When you've finished typing in your program, select Create Build Commands from the Build menu. You'll see the dialog box shown in Figure 2-9.

■ **Figure 2-9**   CreateMake dialog box

```
┌──────────────────────────────────────────────────────────┐
│ ┌─CreateMake Options──────────────────────────────────┐   │
│ │                                                      │   │
│ │  Program Name  │MyProgram│      ┌─ Source Files... ─┐│   │
│ │                                                      │   │
│ │  ┌─Program Type──┐    Creator  │????│                │   │
│ │  │ ◉ Application │       Type  │????│                │   │
│ │  │ ○ Tool        │  Main Entry Point │          │    │   │
│ │  │ ○ Desk Accessory                               │   │
│ │  │ ○ Code Resource │  Resource Type │          │    │   │
│ │  └───────────────┘                                   │   │
│ │                   ☐ Symbolic debugger information     │   │
│ └──────────────────────────────────────────────────────┘   │
│ ┌─Command Line───────────────────────────────────────────┐ │
│ │ createmake MyProgram                                   │ │
│ └────────────────────────────────────────────────────────┘ │
│ ┌─Help──────────────────────────────────┐  ┌─ Cancel ─┐   │
│ │ Create a simple makefile for building │                  │
│ │ an application, tool, or desk          │  ┌CreateMake┐   │
│ │ accessory. The makefile is for use by  │                 │
│ │ the Build menu.                        │                 │
│ └───────────────────────────────────────┘                 │
└──────────────────────────────────────────────────────────┘
```

Type in the program's name (without ".a", ".c", ".cp", or ".p" suffixes) and click a radio button to indicate whether you want to create an application, stand-alone code resource, desk accessory, or MPW tool. When you click the Files button, another dialog box appears, permitting you to select the needed source and library (ending with the ".o" suffix) files. Your program will be linked with these files.

◆ *Note:* It isn't necessary to indicate the standard library files supplied with MPW. Your program will be automatically linked with the appropriate libraries. The reference for CreateMake in Part II explains which standard library files will be used.

The Create Build Commands command in the Build menu runs a script that creates a makefile with the necessary commands for building programs written in assembly language, C, C++, Pascal, Rez, or a combination of languages. This file is given your program's name with the suffix ".make".

◆ *Note:* The Build script uses Make to determine the minimum operations necessary to bring the program up to date. The Build script looks for its build instructions first in *program*.make (for example, Sample.make). If no such file is found, the Build script looks for its instructions in MakeFile.

4. Select a build command from the Build menu.

The four build commands on the Build menus are variations on a theme. (See Chapter 3 for an explanation of each item. A brief explanation appears earlier in this chapter under Step 2 of "Two Easy Steps.") For now, select Full Build. The rotating beach ball cursor appears, indicating that processing has begun. Each step of the build process is displayed on the worksheet as it occurs. Any errors will be displayed also, making it easy to track down a bit of misplaced syntax. When you have fixed the problem, just select Build from the Build menu to quickly rebuild the program. The record of previous builds is left on the worksheet.

See Part II for detailed information on each of the Build menu commands.

Chapter 3 Using the Shell Menus

THIS CHAPTER DESCRIBES THE MENUS AND ASSOCIATED DIALOG BOXES of the Macintosh Programmer's Workshop 3.0 Shell. You can build simple programs by using the Directory, File, and Build menus. (See Chapter 2 for an easy demonstration.) The other menus are used for general editing. More advanced editing capabilities, such as scripted editing and selection specification, are discussed in Chapter 6. ∎

Contents

Features

The MPW Shell provides the following editing features:

- Both menu and command-language editing. The menu commands provide the usual Macintosh interface.

- Selecting text by program syntax. You can double-click any of these paired quotation characters:

  ( )   [ ]   { }   " "   ' '   ` `   / \

  to select everything between the character and its mate. To select text between

  " "   ' '   ` `   / \

  click the *left* quotation character.

- Selection of large sections of text by embedding markers. Marked selections are scriptable; your command files can refer to one or more marked selections. The marker commands, Mark and Unmark, are available from the Mark menu. Basic interactive use of markers is covered later in this chapter. See Chapter 6 for more detailed information on scripting marked selections.

- Complete integration of editing functions with the command interpreter. In the MPW Shell, there is no separation of "command" and "editor" modes. To the Shell, text is text—it is only when you try to directly execute a string of text that the Shell decides whether it is a legitimate command or not.

- Scriptable commands. Because editing commands are part of the command language, you can use them with structured commands and variables to put together scripts that define new editing commands. (See Chapter 6.)

- Regular expressions for matching text patterns. These make possible powerful search-and-replace functions that eliminate the need to make repetitive changes by hand. (See Chapter 6.)

File format

Shell text is saved as a text-only (TEXT) file. The file contains tab and return characters, but no other formatting information. This format is compatible with other applications that create text-only files—for example, the Shell can process MacWrite® files saved with the Text Only option. When you select the Open command, the Shell displays all text-only files in its standard file dialog box, regardless of the file creator.

△ **Important**   From the Finder, you can open a text file created by another application by selecting both the MPW Shell and the text file icons, and then choosing the Open command. △

You can display the invisible characters (spaces, tabs, returns, and all other "control" characters) with the Show Invisibles checkbox in the Format dialog box.

A file's tab setting, font setting, selection, window settings, auto-indent state, invisibles state, and markers are saved with the file in its resource fork.

Menu commands

In general, the menu interface is the familiar Macintosh implementation. There are a few differences and extensions, which are detailed in the following sections. (It's assumed that you are already familiar with standard Macintosh editing techniques.) Many menu commands are scriptable, that is, a command-line form of the command exists (and is described in Part II) that lets you use the menu item noninteractively in a script. Each of these are indicated later in this section.

All menu commands act on the active (that is, the frontmost) window.

Apple menu

Open the "About MPW" menu item to display version information.

File menu

The File menu contains the Shell commands for creating, opening, printing, closing, and saving files.

- **Figure 3-1**   File menu
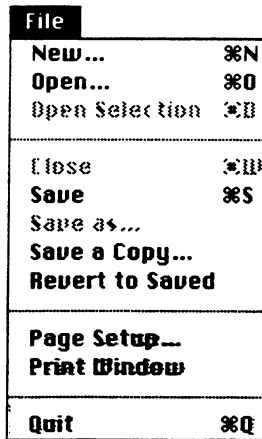
If the Worksheet is the current window, the menu commands Close will appear dimmed, as shown in Figure 3-1. If a tool is executing, all menu commands (except New and Open) appear dimmed.

New

Displays the New dialog box, shown in Figure 3-2. The MPW New dialog box allows you to enter a name and select a directory location for the document. The Command-key equivalent is Command-N. There is also a scriptable New, described in Part II.

- **Figure 3-2**   New dialog box
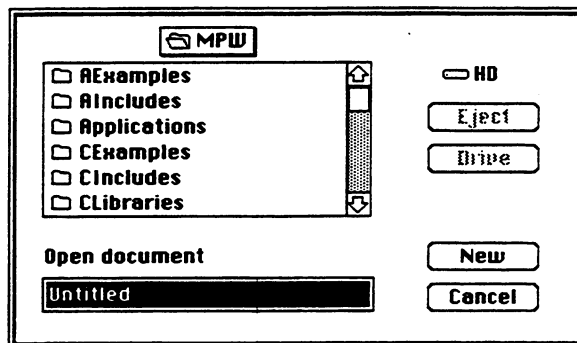
Open

Displays an Open dialog box (similar to that in Figure 3-2) that allows you to open any TEXT file on the disk. When you open a file for the first time, the selection point is at the top of the file. When you open the file again, it reappears in the same state in which it was saved; that is, the previous selection or insertion point is preserved unless the file has been modified outside the editor. The Read Only checkbox is located just below the Open Document box. Check the Read-Only box to open a nonmodifiable copy of the file. The Command-key equivalent is Command-O. There is also a scriptable Open, described in Part II.

◆ *Note:* If you try to open a document that's already open in another window, that window will be brought to the front. Whenever you open a file, it appears in a new window.

Open Selection

If you select a document name within a window, the Open Selection command automatically displays the selected name. This is a useful shortcut when you have already displayed filenames on the screen, with the Files command, for example. You can then select a filename and open a file directly, bypassing the usual Open dialog box. Variable and command substitution occur on the selection. The Command-key equivalent is Command-D.

Close

Closes the active (frontmost) window. The Command-key equivalent is Command-W. There is also a scriptable Close, described in Part II.

Save

Saves the active window under its current name, without closing it. This menu item is dimmed if the contents of the window haven't been modified since it was last saved. The Command-key equivalent is Command-S. There is also a scriptable Save, described in Part II.

Save As

Displays a Save As dialog box, allowing you to change the name and directory location of the active window. Saves the current contents of the window as the Save As file, and allows you to continue editing the new file. The old file is closed without saving, under its original name.

Save a Copy

Saves the current state of the active window to a new file on the disk. You can then continue editing the *old* file.

Revert to Saved

Throws away any changes you have made since you last saved the active window. This menu command is dimmed if the window has not been modified since you last saved. There is also a scriptable Revert, described in Part II.

Page Setup

Displays the standard Page Setup dialog box.

Print Window/Print Selection

Prints either the entire contents of the active window or the selection in the active window. If any text is selected in the active window, that text is printed. If no text is selected, the entire contents of the window (that is, the entire file) are printed.

◆ *Note:* For the Print command to work properly, you must install the printer drivers available on the latest version of the Printer Installation disk. Use the Chooser Desk Accessory from the Apple menu to specify which printer to use. Use the Page Setup dialog box to specify paper size, orientation, and reductions or enlargements.

The Print menu item doesn't display the usual Print dialog box. Instead, you can specify printing parameters by setting the Shell variable {PrintOptions}, described in Chapter 5. Printing options include

■ the number of copies to print

■ which pages to print

■ print quality

■ font

■ font size

■ headings

■ title

■ borders

■ printing the pages in reverse order (for use with the LaserWriter)

See the description of the Print command in Part II for a complete specification of these options, or enter the command Help Print to see a summary.

◆ How Print works

The Print Window menu item executes the Shell command

```
Print {PrintOptions} "{Active}" ≥≥ "{Worksheet}"
```

Print Selection executes the same command with .§ added after the name of the active window. ◆

Quit

Quit returns you to the Finder, first allowing you to save all open files. The Command-key equivalent is Command-Q. There is also a scriptable Quit, described in Part II.

Edit menu

In addition to the usual Macintosh editing commands, the MPW Edit menu (Figure 3-3) contains a few special menu items. See "Editing With the Command Language" in Chapter 5 for more information on using the scriptable forms of the commands on this menu.

■ **Figure 3-3**   Edit menu

```
┌─────────────────────┐
│ Edit                │
├─────────────────────┤
│ Undo          ⌘Z    │
├─────────────────────┤
│ Cut           ⌘K    │
│ Copy          ⌘C    │
│ Paste         ⌘U    │
│ Clear               │
├─────────────────────┤
│ Select All    ⌘A    │
│ Show Clipboard      │
├─────────────────────┤
│ Format...     ⌘Y    │
├─────────────────────┤
│ Align               │
│ Shift Left    ⌘[    │
│ Shift Right   ⌘]    │
└─────────────────────┘
```

Undo

Undoes the most recent changes to *text* in the active window (but *not* changes to resources such as font or tab settings). You can select Undo again to redo changes. The Command-key equivalent is Command-Z. There is also a scriptable Undo, described in Part II.

Cut

Copies the current selection in the active window to the Clipboard and then deletes it from its original location. The Command-key equivalent is Command-X. There is also a scriptable Cut, described in Part II.

Copy

Copies the current selection in the active window to the Clipboard. The Command-key equivalent is Command-C. There is also a scriptable Copy, described in Part II.

Paste

Replaces the contents of the current selection in the active window with the contents of the Clipboard. The Command-key equivalent is Command-V. There is also a scriptable Paste, described in Part II.

Clear

Deletes the current selection in the active window. There is also a scriptable Clear, described in Part II. The keyboard equivalent is the Clear key.

Select All

Selects the entire contents of the active window. The Command-key equivalent is Command-A.

Show Clipboard

Opens a window displaying the contents of the Clipboard, if any.

Format

Displays the Format dialog box offering a selection of fonts and sizes. The Command-key equivalent is Command-Y. This dialog box is shown in Figure 3-4. There is also a scriptable Format, described in Part II.

■ **Figure 3-4**   Dialog box of the Format menu item
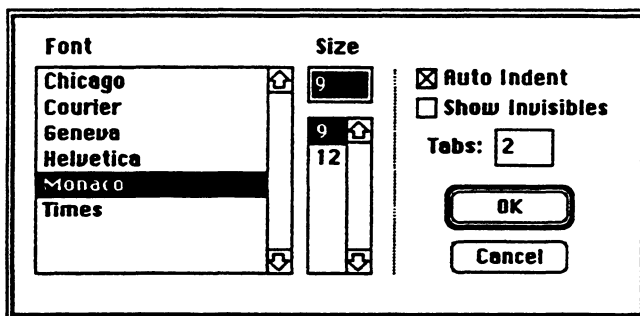
◆ *Note:* Selecting a font and font size affects the *entire* active window, not just the current selection in that window.

Tabs	Sets the number of spaces that a tab character will signify for the active window.
	You can set the default format for a new window by using the Shell variables {Font}, {FontSize}, {Tab}, and {AutoIndent}. These are described in Chapter 5.
Auto Indent	Toggles Auto Indent on and off. When Auto Indent is on, pressing Return lines up text with the previous line. (A check mark indicates that Auto Indent is on.)

Temporary disable feature: To temporarily disable Auto Indent for one line, press Option-Return. That line will begin flush left.

Show Invisibles Displays these invisible characters:

Tab	Δ
Space	◊
Return	¬
All other control characters	¿

The MPW Shell editor ignores any zero-width characters (that is, control characters that do not have a character bitmap) typed from the keyboard. (Usually these are typed by accident.) If you really want a control character in your document, you can enter it in the Key Caps desk accessory and then paste it in your document. To delete control characters that might not be visible, select Show Invisibles from the Format dialog box.

The rest of the dialog box consists of a selection of the fonts installed in your System file. Available font sizes are displayed in the dialog window.

Align

Aligns the currently selected text with the top line of the selection.

Shift Left, Shift Right

These commands move the selected text left or right by one tab stop. You can thus move a block of text while maintaining indentation.

Shift Left	Removes a tab from the beginning of each line. The Command-key equivalent is Command-[.
Shift Right	Adds a tab, or the equivalent number of spaces, to the beginning of each line. The Command-key equivalent is Command-].

If you hold down the Shift key while using these menu items, the selection will be shifted by one space, rather than by one tab.

Find menu

The Find menu contains the routine commands for searching and replacing text. Each of the items in the Find menu is described below.

■ **Figure 3-5**   Find menu

```
┌─────────────┐
│ Find        │
├─────────────┤
│ Find...    ⌘F│
│ Find Same  ⌘G│
│ Find Selection ⌘H│
│ Display Selection│
├─────────────┤
│ Replace... ⌘R│
│ Replace Same ⌘T│
└─────────────┘
```

Find

Displays a Find dialog box and finds the string you specify. By default, the Shell editor searches forward from the current selection in the active window (and does not wrap around). The Command-key equivalent is Command-F. This dialog box is very similar to the Find-and-Replace dialog box described under Figure 3-6; that explanation of the radio controls and check boxes applies to both dialog boxes. There is also a scriptable Find, described in Part II.

Find Same

Repeats the last Find operation, on the active window. The Command-key equivalent is Command-G.

Find Selection

Finds the next occurrence of the current selection in the active window. The Command-key equivalent is Command-H.

Display Selection

Scrolls the current selection in the active window into view.

Replace

Displays the Find-and-Replace dialog box shown in Figure 3-6 and explained there. The Command-key equivalent is Command-R.

Replace Same

Repeats the last Replace operation. The Command-key equivalent is Command-T.

- **Figure 3-6**   Dialog box of the Replace menu item

```
┌─────────────────────────────────────────────────┐
│  ┌───────────────────────────────────────────┐  │
│  │ Find what string?                         │  │
│  │ ┌───────────────────────────────────────┐ │  │
│  │ └───────────────────────────────────────┘ │  │
│  │ Replace with what string?                 │  │
│  │ ┌───────────────────────────────────────┐ │  │
│  │ └───────────────────────────────────────┘ │  │
│  │ ------------------------------------------ │  │
│  │ ⦿ Literal          ☐ Case Sensitive       │  │
│  │ ○ Entire Word      ☐ Search Backwards     │  │
│  │ ○ Selection Expression ☐ Wrap-around Search│  │
│  │ ------------------------------------------ │  │
│  │ ( Replace )  [Replace All]  ( Find )( Cancel )│ │
│  └───────────────────────────────────────────┘  │
└─────────────────────────────────────────────────┘
```
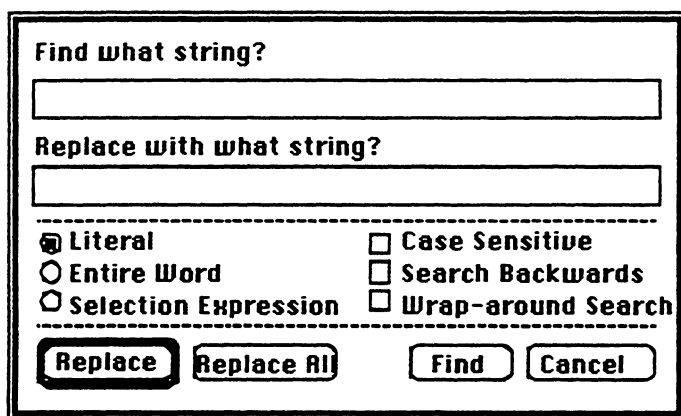
The operation of this dialog box is very similar to that of the Find dialog box, except that selected strings can be located and replaced with a different string throughout a file. Both the Find and the Replace dialog boxes have three radio buttons, offering you one of three options:

Literal	Finds the exact string that you specify, wherever it may appear, even if it is part of other words or expressions.
Entire Word	Finds the specified string only when it occurs as a single word. To the editor, a word is composed of the characters a–z, A–Z, 0–9, and the underscore character (_). (You can change these default values by redefining the Shell variable {WordSet}—see "Predefined Variables" in Chapter 5.)
Selection Expression	Enables the full selection and regular expression syntax, as used with the command language and described in Chapter 6. These expressions allow powerful selection and pattern-matching capabilities that use a special set of metacharacters introduced later in this section.

Any combination of these three check boxes may be selected:

Case Sensitive	Searching is normally case insensitive; selecting this checkbox specifies case-sensitive searching.
Search Backwards	Search backward from the current selection to the beginning of the file. (Normally, searching is forward and stops at the end of the file.)
Wrap-Around Search	Searches forward to the end of file, then wraps around and searches from the beginning of the file to the cursor's location when the search was initiated. (The direction of Search is reversed if Search Backward is also selected.)

These dialog options set the Shell variables {CaseSensitive}, {SearchBackward}, {SearchWrap}, and {SearchType}. You can also use these variables in scripts to set the related options in the dialog boxes. See "Variables Defined in the Startup File" in Chapter 5.

For Find and Find-and-Replace operations, a beep indicates that the string was not found.

◆ **Hints on using Find**

You can reverse the direction of a current search operation by pressing Shift as you select the menu item or click the OK button. The direction is changed for the current search operation only; the settings of the dialog's check box and the {SearchBackward} variable are not affected.

For example, if you are in the middle of a file and you want something above the current cursor position, then hold down the Shift key as you click OK. The search will then proceed backward through the first part of the file.

You might also use the Shift key to make sure that you've found all instances of an item from an arbitrary position in the window. Press Command-G to run Find Same forward. Press Shift-Command-G to run Find Same backward. ◆

Selection expression

When the Find-and-Replace dialog box's "Selection Expression" switch is selected, you can use a special set of expression operators to specify selections and text patterns. This section introduces a commonly used subset of these selection operators. Many more capabilities are available. A full discussion of them can be found in Chapter 6.

Selection by line number: A number given by itself specifies a line number. In Figure 3-7, for example, the command selects line 30 in the active window.

■ **Figure 3-7**   Selection by line number

```
┌─────────────────────────────────────────────────┐
│ ╔═══════════════════════════════════════════════╗ │
│ ║                                               ║ │
│ ║  Find what selection expression?              ║ │
│ ║ ┌───────────────────────────────────────────┐ ║ │
│ ║ │ 30|                                       │ ║ │
│ ║ └───────────────────────────────────────────┘ ║ │
│ ║                                               ║ │
│ ║  ○ Literal            □ Case Sensitive        ║ │
│ ║  ○ Entire Word        □ Search Backwards      ║ │
│ ║  ◉ Selection Expression □ Wrap-Around Search  ║ │
│ ║                                               ║ │
│ ║  ┌──────────┐              ┌──────────┐       ║ │
│ ║  │  Find    │              │ Cancel   │       ║ │
│ ║  └──────────┘              └──────────┘       ║ │
│ ╚═══════════════════════════════════════════════╝ │
└─────────────────────────────────────────────────┘
```
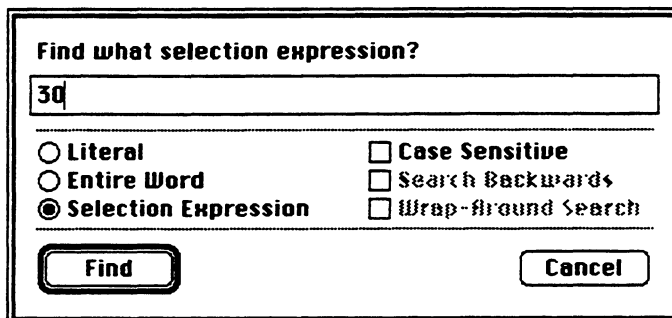
Wildcard operators: The same wildcard operators used in filename generation can also be used to specify text patterns for Find commands:

?	Any single character (other than Return).
≈	Any string of 0 or more characters, that does not contain a Return. (To get the ≈ character, press Option-X.)
[*characterList*]	Any character in the list.
	Note: The brackets must be typed; they don't indicate an optional syntax element.
[¬*characterList*]	Any character *not* in the list. (To get the ¬ character, press Option-L.)

These pattern-matching operators are part of a larger set called **regular expression operators,** used to define searches and other scripted operations. A regular expression consists of literal characters and/or regular expression operators, and it must be enclosed in slashes (/.../). Figure 3-8 shows an example.

■ **Figure 3-8**   Example of a regular expression
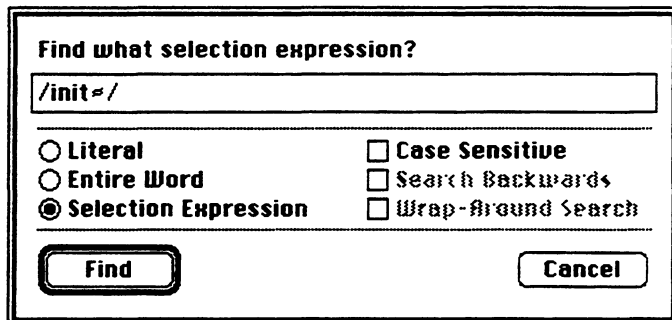
The command shown in Figure 3-8 finds and selects any string that begins with "init" and is followed by any characters other than a return or a space. Figure 3-9 shows the result of this command.

■ **Figure 3-9** Text selected with the Find command


```
HD:MPW:Examples:PExamples:Sample.p

{$S Main}
BEGIN
    UnloadSeg(@_DataInit);     {note that _DataInit must not be in Main!}
    ForceEnvirons;             {check for some basic requirements; exits i
    MaxApplZone;               {expand the heap so code segments load at t

    Initialize;                {initialize the program}
    UnloadSeg(@Initialize);    {note that Initialize must not be in Main!}

    EventLoop;                 {call the main event loop}
END.

        MPW Shell
```

As mentioned, many additional Find-and-Replace capabilities are available. (See Chapter 6.) In the command language, the Find-and-Replace functions are performed by the Find-and-Replace commands. There's also a tool named Search (described in Part II) that can search through a list of files for the occurrence of any text pattern.

Mark menu

A **marker** is a text selection that has been given a name. Markers are useful for navigating within a window, and they can simplify many selection expressions. The upper part of the Mark menu contains the commands Mark and Unmark and the lower part lists all existing markers. (By the way, when you first start MPW 3.0, you'll notice that this area of the Mark menu contains a list of MPW commands that have been marked in order to display them conveniently in a menu. You can unmark them if you prefer.) To jump to the location of a marker, you simply choose the name of the marker you want from the Mark menu, shown in Figure 3-10.

Markers can be created and used both interactively, via the Mark menu, and programmatically, via the Shell commands Mark, Unmark, and Markers. For a detailed discussion of the syntax, characteristics, and programmatic use of markers, see Chapter 6 and Part II.

■ **Figure 3-10**   Mark menu

```
Mark
 Mark...   ⌘M
 Unmark...

 Commando
 Examples
 Help
 Alias
 Catenate
 Clear
 Close
 Copy
 Count
 Cut
 Date
 Delete
 Duplicate
 Echo
 Eject
  ▼
```
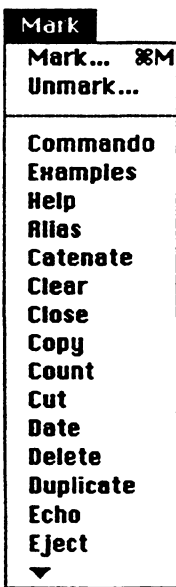
Mark

To create a new marker interactively, first select the text you want to mark, then choose "Mark" from the Mark menu. A dialog box like that in Figure 3-11 appears, asking for the name you want the marker to have. The editable text field in the Mark dialog box is initialized to the first word (that is, whatever you would select by a double click) in the selection. If you click Cancel in the dialog box, the selection is unchanged and no new marker is created. If you click OK, a new marker is created with the specified name and the new marker's name is added to the list of marker names displayed by the Mark menu.

■ **Figure 3-11**   Mark dialog box

```
 Mark the selection with what name?
 [                                    ]

  [  OK  ]            [ Cancel ]
```
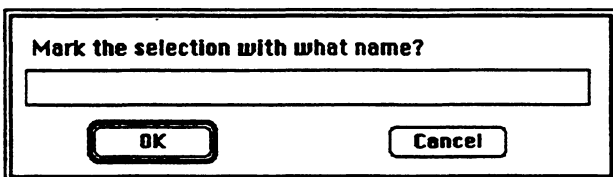
If you try to create a new marker using the name of an already existing marker, a dialog box will appear, giving you the chance either to delete the old marker and add the new (OK), or to forget about adding the new marker (Cancel).

◆ **Hints on using Mark**

Markers are very handy for quick navigation through source files. You may want to mark important data declarations and all procedures so that you can quickly jump to any procedure by selecting its marker. Markers are listed according to their position in the file. ◆

Unmark

If you choose the Unmark menu item from the Mark menu, you'll see a dialog box (Figure 3-12) that contains a list of currently defined markers and the two buttons Delete and Cancel. If a marker is currently selected, its name is highlighted in the marker list. You can select any number of marker names from the list. If you click Delete, every marker selected in the list is deleted. If you click Cancel, the selection remains unchanged and no markers are deleted.

■ **Figure 3-12**   Unmark dialog box
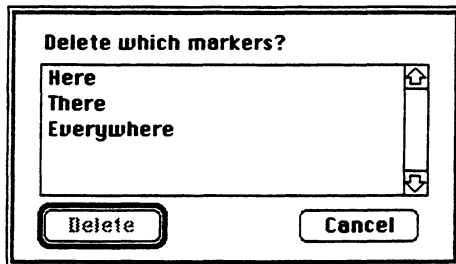
Window menu

The upper portion of the Window menu contains the two commands Tile Windows and Stack Windows; the middle area lists all open windows, as shown in Figure 3-13. The lower area of the Window menu lists any open Projector windows.

- **Figure 3-13** Window menu

```
 Window
 Tile Windows
 Stack Windows

 ✓HD :MPW:Worksheet
```

Tile Windows

Use this command to arrange windows in a tile pattern on the screen so that each window's contents are visible. To include the Worksheet in the tiling, press the Option key as you select Tile Windows.

Stack Windows

Use this command to arrange windows in a diagonally staggered pattern on your screen. This is the "open file folder" way to see several windows at once. To include the Worksheet in the stacking, press the Option key as you select Stack Windows.

Customizing window commands

The Tile Windows and Stack Windows menu commands execute the Shell commands:

```
TileWindows {TileOptions} >> "{WorkSheet}"
StackWindows {StackOptions} >> "{WorkSheet}"
```

You may customize the operations of tiling and stacking by using the Shell variables {TileOptions} and {StackOptions}. Options include

- which windows to tile
- including the Worksheet (without pressing the Option key)
- horizontal or vertical tiling
- spacing between stacked windows
- specifying a rectangle in which to tile or stack windows

List of open windows

The remainder of the menu lists all open windows in the order in which they were opened. The full pathname is listed. To bring any window to the front, select that window from the list.

Selecting a window from the menu brings that window to the front, that is, superimposes it over anything else on your display. A check indicates that the window is currently the "active" window, that is, the frontmost. A bullet (•) indicates that the window is the "target" window, that is, the second to the front. Underlining indicates that a window contains changes that have not yet been saved.

Project menu

The Project menu, shown in Figure 3-14, puts three of the most often used Projector commands at your fingertips. Of course, you can modify this menu to add the rest of Projector's commands or eliminate the menu altogether if you don't use it.

The three menu items on the Project menu are briefly described here. For an introduction to the basics of using these functions, see "Projector Windows" in Chapter 4. For a detailed explanation of the MPW project-management system, see Chapter 7.

■ **Figure 3-14**   Project menu

```
┌──────────────┐
│ Project      │
├──────────────┤
│ New Project... │
│ Check In...   │
│ Check Out...  │
└──────────────┘
```

New Project

The New Project dialog box appears as shown in Figure 3-15. Use this dialog box to create a unique new project or subproject. You can use the Comment text frame to briefly explain the purpose of the project or subproject. Projector automatically adds your user name as the project's creator.

■ **Figure 3-15** New Project dialog box
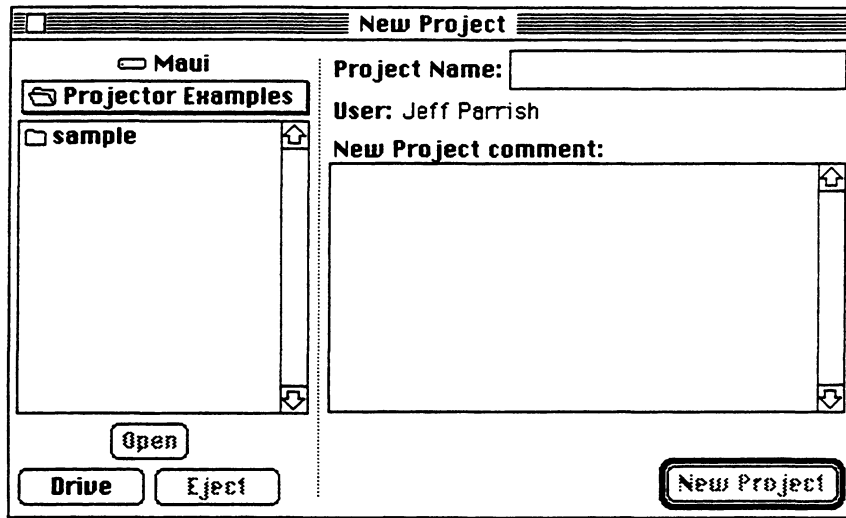
Check In

The Check In dialog box appears as shown in Figure 3-16. After checking out and modifying a file, you will routinely use this dialog box to check the file back in to Projector.

■ **Figure 3-16** Check In dialog
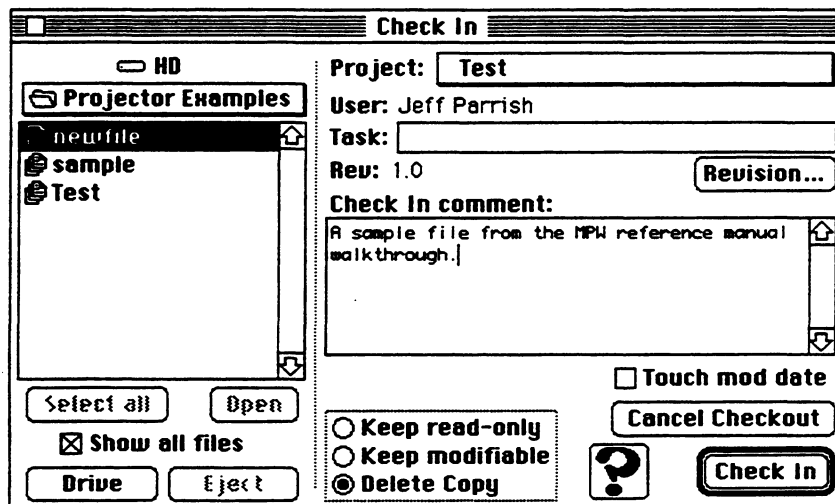
Click the Question Mark button to display information about the project, a project file, or a specific revision of a project file. See Chapter 7 for more information.

Check Out

The Check Out dialog box appears as shown in Figure 3-17. You'll routinely use this dialog box to select a project for use and then to check out a project file you want to modify. The date, time, and user name of the checked-out file are recorded; no one else can modify the same revision of a file at the same time.

■ **Figure 3-17**   Check Out dialog box
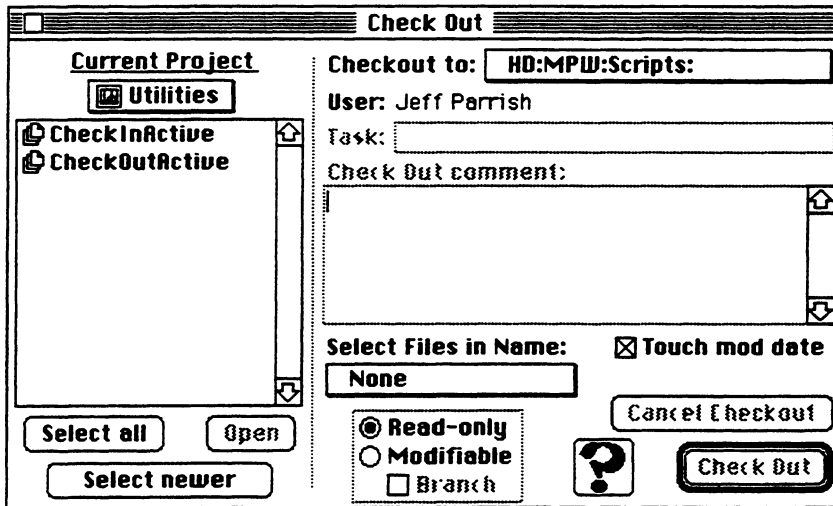
Click the Question Mark button to display information about the project, a project file, or a specific revision of a project file. See Chapter 7 for more information.

Directory menu

The Directory menu, shown in Figure 3-18, lets you display and easily change the default (current) directory. The Directory menu is implemented by the scripts DirectoryMenu and SetDirectory, which you can modify to suit your own needs.

■ **Figure 3-18** Directory menu

```
┌────────────────────────────┐
│       ▓▓Directory▓▓         │
├────────────────────────────┤
│ Show Directory             │
│ Set Directory...           │
├────────────────────────────┤
│ HD2:MPW:Examples:AExamples:│
│ HD2:MPW:Examples:CExamples:│
│ HD2:MPW:Examples:CPlusExamples:│
│ HD2:MPW:Examples:Examples: │
│ HD2:MPW:Examples:PExamples:│
│ HD2:MPW:Examples:Projector Examples:│
│ HD2:MPW:                   │
└────────────────────────────┘
```
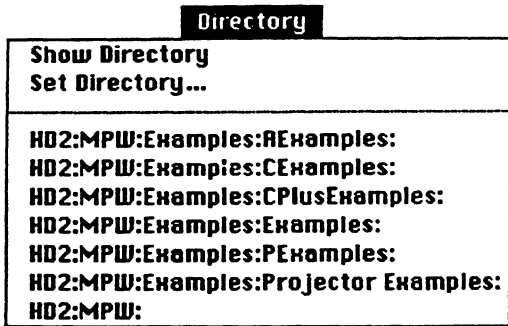
Show Directory

An alert box displays the name of the current default directory.

Set Directory

When you select this menu command the Set Directory dialog box (Figure 3-19) appears, providing interactive selection of the default directory. Your selection is then added to the Directory menu.

■ **Figure 3-19** Dialog box of the Set Directory menu item

```
┌──────────────────────────────────────────┐
│  ╭──────────────────────────╮             │
│  │ Select Current Directory: │            │
│  ╰──────────────────────────╯             │
│        ┌─────────┐                        │
│        │ 🖴 MPW   │                        │
│    ┌─────────────────────┐                │
│    │ 🗀 Examples       ▲ │    ▭ HD2        │
│    │ 🗀 Interfaces     ▒ │                 │
│    │ 🗀 Libraries      ▒ │   ╭──────────╮  │
│    │ 🗀 MPW Demo       ▒ │   │  Eject   │  │
│    │ 🗀 ROM Maps       ▒ │   ╰──────────╯  │
│    │ 🗀 Scripts        ▒ │   ╭──────────╮  │
│    │ 🗀 Tools          ▒ │   │  Drive   │  │
│    │                  ▒ │   ╰──────────╯  │
│    │                  ▼ │   ╭──────────╮  │
│    └─────────────────────┘   │   Open   │  │
│                              ╰──────────╯  │
│                              ╭──────────╮  │
│                              │Directory │  │
│                              ╰──────────╯  │
│                              ╭──────────╮  │
│                              │  Cancel  │  │
│                              ╰──────────╯  │
└──────────────────────────────────────────┘
```
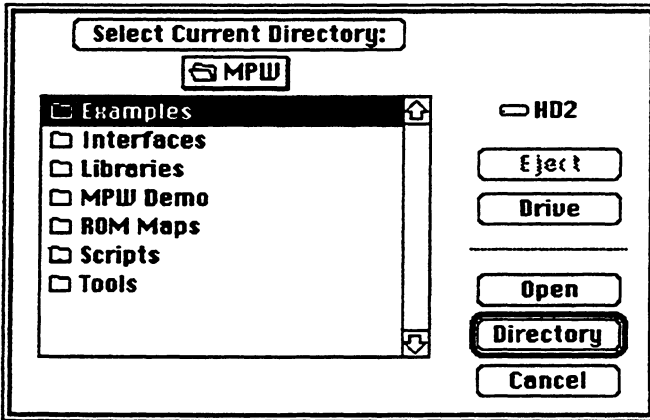
List of directory names

Selecting a directory name makes this directory the new default directory.

As you select various default directories, using either the Set Directory menu command or the SetDirectory command, each is added as a separate menu command to make it easy to return to that directory in the future. The UserStartup script creates menu items for each of the Examples folders in the MPW directory, and for the default directory at the time the UserStartup script is run. You can add your own favorite directories by modifying UserStartup.

▲ Warning   Directory names should not contain any of these special characters:

- ; ^ ! < / (

These characters have special meanings when they appear as menu items. ▲

Build menu

The Build menu, shown in Figure 3-20, has two primary purposes. The first purpose of the Build menu is to create a makefile containing the commands needed to build a program. The command Create Build Commands, which is listed first on the menu, creates the makefile *program*.make (using the name of your program). If you have not used this command—that is, if *program*.make does not exist—then MPW uses the file Makefile.

The second purpose of the Build menu is either to build a specified program or to display the commands needed to do the build. When you select one of the remaining commands on the menu—Build, Full Build, Show Build, and Show Full Build Commands—a dialog box appears asking for the name of the program that you want to build.

Use of the Build menu is demonstrated in Chapter 2, "Building a Program: An Introduction."

■ **Figure 3-20**   Build menu

Build
Create Build Commands...
Build...　　　　　　　　　　　**⌘B**
Full Build...
Show Build Commands...
Show Full Build Commands...
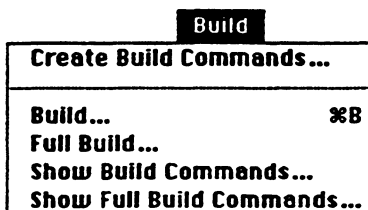
Create Build Commands

Use this item to create a makefile containing the build commands for a specified program. When you click Create Build Commands, the CreateMake dialog box appears. (See Figure 3-21.) You can then enter the program name and select its type (that is, Application, Tool, or Desk Accessory). Make sure that you do *not* include any of the following four suffixes to the program name:

.a   .c   .p   .cp

Click the Files button to select the program's source and library files. (MPW libraries are automatically linked; certain special libraries you may require might not be automatically linked. See CreateMake in Part II.) If the program's name is *program*, a new makefile, called "*program*.make", is created. The makefile will contain simple build commands from the program. (See Chapter 9 for more information on Make.)

Be sure to run Create Build Commands whenever you create additional source or library files for a program. Answering the CreateMake dialog box generates a new set of rules in *program*.make that includes the new source files.

■ **Figure 3-21**   CreateMake dialog box
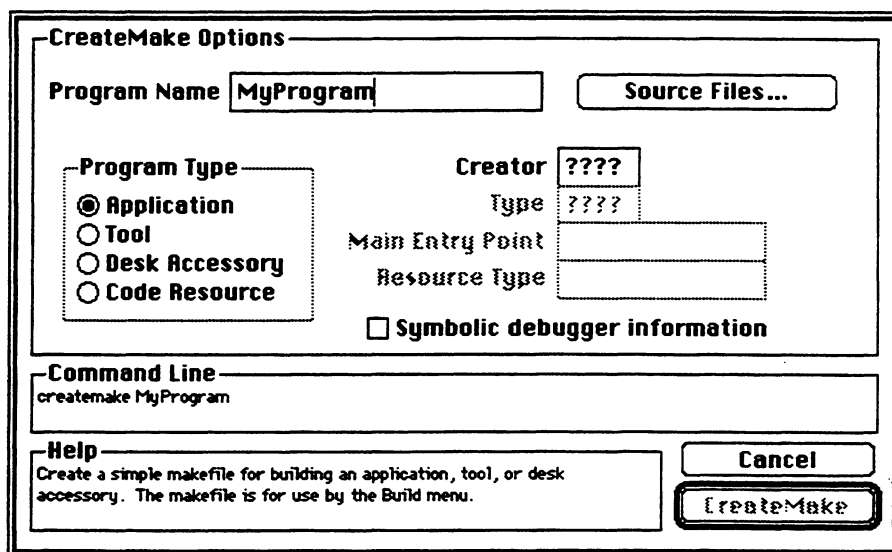
When you select one of the following four Build items from the Build menu, a dialog box appears (as shown in Figure 3-22), asking for the name of the program you want to build.

■ **Figure 3-22**   Program Name dialog box
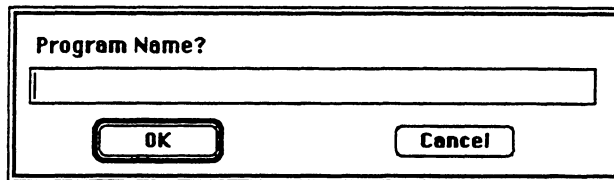
Type the name and click OK. The build option you have selected will proceed, displaying on the Worksheet each command needed to build the program as it is used, along with any error messages. Each of these four Build menu items uses the MPW tool Make to determine which operations are necessary to build the program.

Build

The program is built automatically, but only files that have been modified since you last built the program will be compiled. Use this item to save time. The Command-key equivalent is Command-B.

Full Build

The program is completely built, ignoring any object files or intermediate files that may exist from a previous build.

Show Build Commands

The commands needed to build the program (for just those files affected by modifications since the last build) are displayed on the worksheet, but not executed. You can then select any or all of the commands—or modify them—and press Enter to execute them.

Show Full Build Commands

All the commands needed to completely rebuild the program (whether modified since the last build or not) are displayed on the worksheet, but not executed. This is a convenient way to see all of the commands used in building the program you've selected.

The Makefile "*program*.make" is created by the Create Build Commands menu item (described previously in this chapter). If you have not used this item—that is, if *program*.make doesn't exist—MPW will use the file MakeFile.

User-defined menus

You can define your own menu commands with the AddMenu command, described at the end of Chapter 5. These commands can be appended to existing menus, or you can create new menus. In fact, the Projector, Directory, and Build menus have been created by using AddMenu. You may add to, change, or delete these menus to suit your individual needs.

Chapter 4 Using MPW: The Basics

THIS CHAPTER INTRODUCES THE BASIC CONVENTIONS FOR MANIPULATING FILES, editing text, executing commands, and responding to dialogs in MPW 3.0. You can easily enter all commands, command options, and parameters by using the menus and dialogs. The basics for directly typing commands in any window are also introduced. A full discussion of command scripting can be found in Chapter 5. For an introduction to building a simple program, using examples contained in the Examples folder, see Chapter 2. Chapter 3 introduces the menus and their contents. Chapter 7 presents the dialogs and complete information on Projector, the project management system. ■

Contents

Editing

Basic editing functions are available as menu commands. You can open a file by using the Open command, or by selecting its name on the screen and choosing the Open Selection command (Command-D) from the File menu. You can select and edit text with the usual Macintosh editing techniques, using menu commands to cut, copy, and paste selected text. The menu commands are described in Chapter 3.

You enter and edit command lines in a window exactly the same way you enter plain text. You can select any stretch of text and press Enter to send the selection to MPW's command interpreter for execution.

Editing with MPW is unique in that most menu functions are duplicated in the Shell command language. Editing and other command-language functions are fully integrated— you enter and execute editing commands just like any other commands. Editing commands are entered in the **active window** (the frontmost window), but they act on text in the **target window** (the second window from the front), or another window that you explicitly name. The command language lets you produce scripts of editing commands: You can save any series of commands as a normal text file and execute the file by simply entering the filename. Command-language editing is discussed further in "Editing With the Command Language" in Chapter 5.

For an explanation of selections, markers, and pattern matching with regular expressions, see Chapter 6, "Advanced Editing."

Entering commands

All MPW commands and their options can be selected from menus and dialog boxes. Generally, this interactive method of command selection is the easiest. You can immediately execute commands selected from menus and dialog boxes, or you can use the dialog boxes to compose complex command lines that can then be copied to a script.

The dialog boxes for MPW commands are generated by the Commando user interface (described in the last section of this chapter). Besides the usual Macintosh dialog boxes, Commando provides several new forms and controls to handle the special requirements of MPW tools. For example, dialogs for commands with many options may have several nested dialog boxes. Which dialog boxes are actually displayed may vary according to dependency relations between the particular options you may have selected. Some of the specialized dialog controls are introduced at the end of this chapter. Other unique dialog boxes are shown in Part II of this reference, with their respective commands. A detailed discussion of all the elements of Commando dialogs can be found in Chapter 13, which explains how to create a Commando interface for your own tools and scripts.

Of course, you can always type commands directly in any window as a series of words separated by spaces or tabs. (See below.)

Typing commands in a window

By default, command output and any error messages appear in the window immediately below the executed command line. Commands are not case sensitive. You can have multiple open files, and you can enter commands in any window.

The simplest commands consist of the command name only. For example, type the command

```
Date
```

and press the Enter key (without pressing Return first—that is, the insertion point must be on the same line as the command when you press Enter). This command outputs the date and time:

```
Tuesday, January 15, 1989 7:12:00 AM
```

Commands can have options. For example,

```
Date -d
```

The -d option tells the Date command to list the date only,

```
Tuesday, January 15, 1989
```

Commands typed into an open file are referred to as **standard input**. Output produced by most commands is sent to an open file called **standard output**, which is normally connected to the window in which the command was entered. Any window that is used to enter standard input and display standard output is referred to as the **console**.

Most commands read from standard input, write their output to standard output, and write error messages to diagnostic output. By default, standard input refers to text that is selected and entered while the tool is running. Standard output and diagnostic output appear following the commands. (These input and output defaults can be changed using I/O redirection. See Chapter 5 for details.)

◆ **Using the Alias command**

You may get tired of typing the entire command name for frequently used commands such as Directory. However, you can easily define your own alternative names with the Alias command. For example, after executing this command,

```
Alias dir Directory
```

you can execute the Directory command by entering the new command name:

```
dir
```

To make an alias definition part of the Shell's standard startup procedure, place the definition in the file UserStartup. See Chapter 5, "The Startup and UserStartup Files." ◆

The Enter key

The Enter key serves as a "do it" button, causing commands to be executed. You can type commands in any window and press the Enter key to execute the command line. You can also select command text that is already on the screen and press the Enter key to execute the selected text. Clicking on the status panel, located at the lower left of a window, has the same effect as pressing the Enter key. Pressing Command-Return also has the same effect as presing the Enter key.

△ **Important**   When no text is selected, the entire line is executed the moment the Enter key is pressed, regardless of where the insertion point is on the line. △

Executing several commands at once

By selecting several lines of command text and then pressing Enter, you can execute any number of commands with one stroke. An example is shown in Figure 4-1.

■ **Figure 4-1**   Pressing Enter to execute selected text
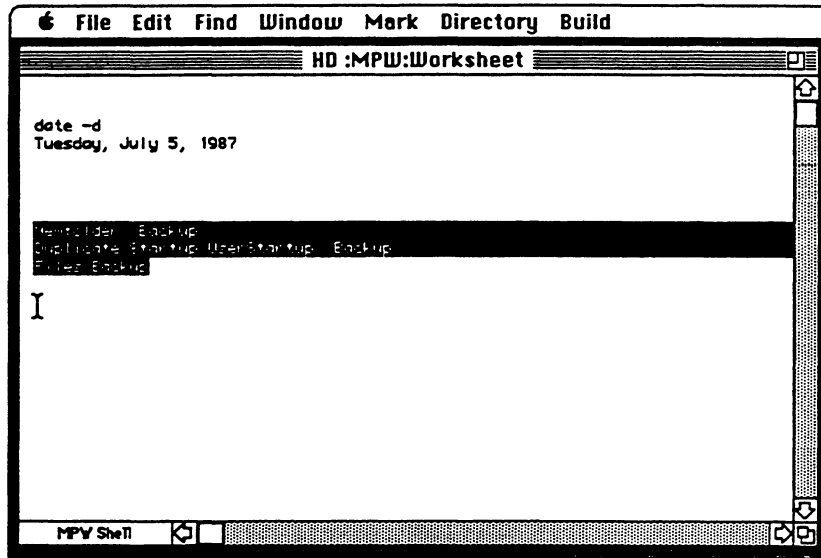
In Figure 4-1, executing the selected text would first make a new folder (directory) named Backup, then copy the files Startup and UserStartup into Backup, and then list all of the files in Backup. (Each of these commands, and the pathname syntax, is described in the sections that follow.)

You can also directly execute text files that contain other commands simply by entering the filename of the script. Executing a script has the same effect as selecting the commands in an open window and pressing Enter—the only difference is the scope of variable and alias definitions (discussed in Chapter 5).

Terminating a command

To terminate a command while it's executing, press Command-period, the standard Macintosh command for this purpose.

△ **Important** Many commands (including Asm, C, and Pascal) normally take their input from a file; however, if no file is specified, they will begin reading from the console (that is, from the window where the command was entered: "standard input"). If the Shell appears not to be listening to the commands you are entering, it probably isn't: The currently executing command (shown in the active window's status panel) may be reading the text that you enter. If a program is reading from standard input, you can press Command-Enter (or Command-Shift-Return) to indicate end-of-file and terminate input. (See "Terminating Input With Command-Enter" in Chapter 5.) △

The Help command

The Help command displays summary information for commands. For example, to display a description of the Files (list files) command and its options, type the command

```
Help Files
```

and press the Enter key. You'll see the following syntax description:

```
Files [option…] [name…]  > fileList
    -c creator           # list only files with this creator
    -d                   # list only directories
    -f                   # list full pathnames
    -i                   # treat all arguments as files
    -l                   # long format (type, creator, size, dates, etc.)
    -m columns           # n column format, where n = columns
    -n                   # don't print header in long or extended format
    -o                   # omit directory headers
    -q                   # don't quote filenames with special characters
    -r                   # recursively list subdirectories
    -s                   # suppress the listing of directories
    -t type              # list only files of this type
    -x format            # extended format--fields specified by format
```

```
Note: The following characters can specify the format
      a    Flag attributes
      b    Logical size, in bytes, of the datafork
      r    Logical size, in bytes, of the resource fork
      c    Creator of File ("Fldr" for folders)
      d    Creation date
      k    Physical size, in kilobytes, of both forks
      m    Modification date
      t    Type
      o    Owner (only for folders on a file server)
      g    Group (only for folders on a file server)
      p    Privileges (only for folders on a file server)
```

◆ *Note:* In Help texts, the brackets are a syntax element indicating that a parameter is optional. An ellipsis (...) indicates that the preceding item may be repeated. (Note that this use of the ellipsis is a syntax convention only for Help text and documentation; an ellipsis character (Option-Semicolon) in an actual command line invokes the command's Commando dialog.) See the section "Syntax Notation" at the end of the Introduction to this reference. The number sign (#) is the MPW comment character.

You can directly edit and execute the text on the screen. For example, assuming that your current directory is {MPW}, you can edit the above text as follows:

1. Use the mouse to select [option...] and [name...]; replace them with the option -l and the directory name Scripts.

2. Remove the output specification > fileList.

The result is a command that will list the files in directory Scripts, in long format:

```
Files -l Scripts
```

(Scripts is the directory containing various MPW scripts; the -l option generates "long" output.) Press Enter to execute the command. Directory information appears immediately following the command.

You can also use the Help command to display additional summary information, including

■ an annotated list of all MPW commands

■ an annotated list of the characters that have special meanings to the MPW Shell

■ descriptions of the syntax of expressions, selections, and text patterns

■ a summary of MPW Shell shortcuts

■ a summary of predefined MPW Shell variables

■ a summary of Projector, the project management system

For general information about Help, execute the Help command with no parameters:

```
Help
```

This command displays the information shown in Figure 4-2.

■ **Figure 4-2**   Help summaries

```
┌═══════════════════ HD:MPW:Worksheet ═══════════════════┐
│ Help                                                  ⇧│
│ MPW 3.0 Help Summaries                                 │
│                                                        │
│    Help summaries are available for each of the MPW commands. │
│    To see the list of commands enter "Help Commands". In addition, │
│    brief descriptions of Expressions, Patterns, Selections, Characters, │
│    Shortcuts, Variables, and Projector are also included. │
│                                                        │
│    To see Help summaries, Enter a command such as      │
│                                                        │
│    Help commandName    * information about commandName  │
│    Help Commands       * a list of commands            │
│    Help Expressions    * summary of expressions        │
│    Help Patterns       * summary of patterns (regular expressions) │
│    Help Selections     * summary of selections         │
│    Help Characters     * summary of MPW Shell special characters │
│    Help Shortcuts      * summary of MPW Shell shortcuts │
│    Help Variables      * summary of the standard MPW shell variables │
│    Help Projector      * summary of Projector, a project/source control system │
│                                                        │
│    Copyright Apple Computer, Inc. 1986-1988            │
│    All rights reserved.                               ⇩│
├──────────────┬─┬───────────────────────────────────────┤
│   MPW Shell  │◁│                                      ⇨⊡│
└──────────────┴─┴───────────────────────────────────────┘
```

You can directly execute the Help commands given in the "Help Summaries" list.

◆ *Note:* The MPW Help file should be in the same directory as the MPW Shell or in the System folder.

File-management commands

The MPW Shell lets you manipulate files without returning to the Finder. Table 4-1 introduces the most commonly used file-management commands.

◆ *Note:* The descriptions in the table omit some of the command options that are available. For complete descriptions, see Part II.

■ **Table 4-1** Basic file-management commands

Command	Description
Backup [*option*] -from folder -to folder [*file...*]	Copy files in source folder to destination folder based on modification date. This is useful when you maintain an identical backup folder on a separate disk.
Catenate [*file...*]	Read the data fork of each file and write it to standard output. (By default, standard output is to the active window, immediately after the command.)
Close [*option*] [-a l *window...*]	Close windows.
Delete *name...*	Delete file or directory *name*. If *name* is a directory, all of its contents are deleted.
Directory *directory*	Set the default directory to *directory*. Directory with no parameters writes the pathname of the current directory.
Duplicate *name... targetName*	Duplicate file or directory *name* to file or directory *targetName*.
Exists *name...*	Determine the existence of file or directory *name*.
Files [*name...*]	List names of directories and files. Options allow you to include various attributes in the listing.
GetFileName [*option...*] [*pathname*]	Display a standard file dialog box.
Mount *drive...*	Mount volumes.
Move *name... targetName*	Move file or directory *name* to *targetName*.
New [*name...*]	Open a new window.
Newer [*option...*] *name... target*	Compare modification dates between files *name* and *target*. List files newer than *target*.
NewFolder *name...*	Create the new directory *name*.
Open [*option*] [*names...*]	Open a window.
Rename *name1 name2*	Rename File or Directory *name1* to *name2*.
Revert	Revert window to previous saved state.
Save [-a l *window...*]	Save windows.
SetDirectory *directory*	Set the default directory.
SetFile [*option...*] *file...*	Set file attributes.

(Continued)

■ **Table 4-1** (Continued) Basic file-management commands

Command	Description
SetPrivilege [*option...*] *folder...*	Set access privileges for folders on the file server.
SetVersion [*option ...*] *file*	Independently maintain the version and revision numbers as a resource in the application or tool. Optionally, update a version and revision string in a source file.
Target *name*	Make a window the target window.
Volumes [*name...*]	List mounted volumes.
WhereIs [*option...*] *pattern*	Find all files that have a partial pathname *pattern*, in any level of any directories.
Which [*command*]	Determine, for the specified *command*, which existing aliases, Shell built-in commands, and commands accessed via the Shell variable {*Commands*} will be executed when *command* is entered.
Windows	List open windows.

File and window names

In the MPW, files and windows are specified in the same way. When a name is passed as a parameter to a command, the system looks first for an open window with that name; if no window is found, it looks for a file on the disk.

The following rules apply to naming:

■ Names are not case sensitive.

■ A single component (file or directory name) of an HFS pathname is limited to 31 characters.

■ Any character except a colon (:) may be used in a file or directory name. (Colons separate elements in a pathname.)

It's best to avoid using spaces and special characters in filenames. When using filenames that contain spaces, you'll need to **quote** them so that they won't be interpreted as individual words in the command language—for example, you would need to specify the name "System Folder" as follows:

```
Files "HD:System Folder"
```

For the rules concerning quoting, see "Quoting Special Characters" in Chapter 5.

Selection specifications

Commands that take filenames for parameters can also act on the **current selection** in a window. The current selection character, § (Option-6), represents the currently selected text in a window. There are two ways to use this character:

§   Currently selected text in the target window. (The target window is the second window from the front, as explained in Chapter 1.)

name.§   Currently selected text in window *name*.

For example, the Count command counts lines and/or characters in a file. The command

```
Count -l Sample.a.§
```

counts the lines within the current selection in the window Sample.a.

The current selection is explained more fully in "Editing With the Command Language" in Chapter 5.

◆ *Note:* The MPW Shell uses a number of special characters (like §) from the extended character set. These characters are fully listed in Appendix C.

Directories and pathnames

With the hierarchical file system (HFS), specifying a filename alone is often not enough to identify a file—you frequently need to specify a pathname. (See Figure 4-3 for a sample HFS structure.) A **full pathname** is specified as follows:

volume : [*directory* : ...] *filename*

A full pathname contains at least one colon (:), but cannot begin with a colon. An example of a full pathname is

```
"HD:MPW:MPW Shell"
```

(The quotation marks are required because the filename "MPW Shell" contains a space.)

■ **Figure 4-3**   Hierarchical directory structure
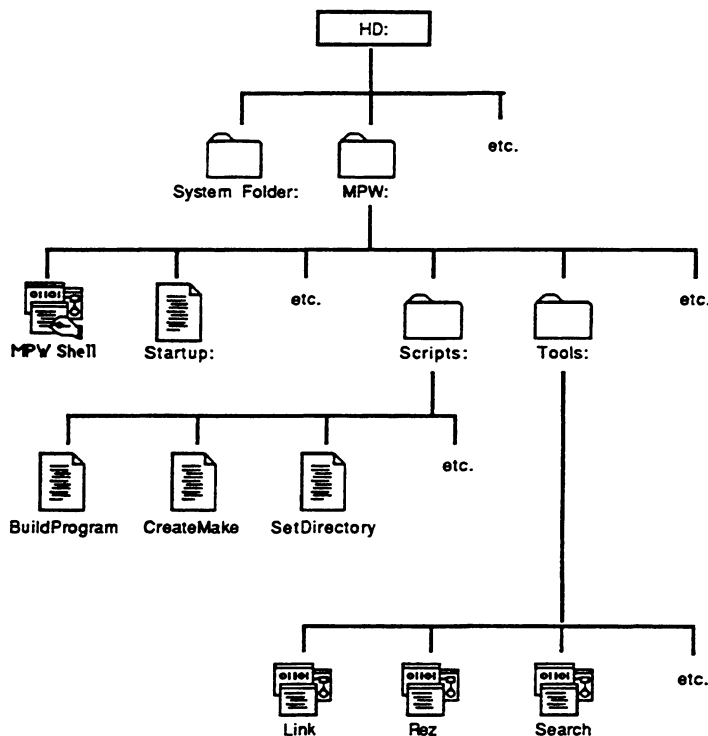
A partial pathname is usually all you'll need to specify. When HFS encounters a partial pathname, it begins the path at the current default directory. Any name that contains no colons or begins with a colon is considered a partial pathname. A partial pathname that contains no colons is a **leafname.** For example, the name

```
:AExamples
```

is taken as a partial pathname. However, the name

```
MPW:
```

is taken to be a *full* pathname (that is, a volume name only), rather than meaning the directory HD:MPW. (When in doubt, you can always specify the full pathname for a file or command.)

Double colons (::) in a pathname specify the current directory's parent directory; triple colons specify the "grandparent" directory (two levels up), and so on. See the chapter "File Manager" in Volume IV of *Inside Macintosh* for more information on HFS conventions.

◆ *Note:* Notice that there's no single "root" directory—each volume name (that is, disk name) is a separate starting point for a directory tree.

You can use the Files command to list the names of files and directories. For example, the command

```
Files HD:MPW:
```

might display the following:

```
:Examples:
:Interfaces:
:Libraries:
:ROM Maps:
:Scripts:
:Tools:
'MPW Shell'
MPW.Help
Quit
Resume
Startup
Suspend
SysErrs.Err
UserStartup
Worksheet
```
...and so on

In the output of the Files command, the names that begin and end with colons are directory names, and the other names are filenames. All of these names are partial pathnames—in this case, "HD:MPW" forms the beginning of each pathname. Also note that filenames containing special characters are quoted.

Command search path

When you enter a command name (that is, a leafname), the Shell searches for the command in the directories listed in the Shell variable {Commands}. As described in Chapter 5, this search path is initially set to

```
:(the current directory)
HD:MPW:Tools:,
HD:MPW:Scripts:,
HD:MPW:Applications:,
```

This means that when you type any command the Shell first assumes that you want to execute a tool; if it can't find the tool, it then assumes that you want a script; if it can't find the script, it then assumes that you want an application. If your frequency of use is different, you can change the search path to improve the Shell's performance. (See Chapter 5.)

Changing directories

You can change the default directory with the Directory command. Assuming you have a hard disk named HD, you could change the default directory to the directory Examples in the MPW folder with the command

```
Directory HD:MPW:Examples
```

Like most commands, Directory runs *silently*—that is, it generates output only if an error occurs. To verify that you have set the appropriate directory, enter the Directory command with no parameters:

```
Directory
```

This command displays the current or default directory.

Remember that to specify a pathname containing spaces or other special characters, you must surround it with single or double quotation marks. (See Chapter 5 for rules on quotation.)

If you specify a directory whose name is a leafname, the Directory command searches the directories listed in the Shell variable {DirectoryPath}. If the variable is undefined, then the command looks in the current directory.

◆ Using the {DirectoryPath}

Here's an easy way to move quickly between directories on different branches. Suppose you have a directory structure like that shown in Figure 4-3, with a DirectoryPath of

```
":,HD:MPW:"
```

Now, if you happened to be in the System folder, you could set your directory to Tools with this command:

```
directory Tools
```

Because this command specifies only a leafname, the Tools directory is looked for first in ":" (where it is not found) and then in HD:MPW (where it is found). The directory is then set to HD:MPW:Tools. ◆

Pathname variables

One way of specifying a pathname is by using Shell variables. For example, the Shell variable {MPW}, defined in the Startup file, expands to form the full pathname for the MPW folder, in this case "HD:MPW:" (assuming that the MPW folder is at the top level). Thus, the Directory command could be entered as

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

In this particular case, the quotation marks aren't necessary. If you adopt the practice of never using spaces or other special characters in a pathname, you don't need to bother with quotation marks. On the other hand, if you sometimes use spaces or other special characters in a pathname, it would be a good idea to use quotation marks whenever variables are included in a pathname.

You can use the Set command to define and redefine variables, as described in Chapter 5. To see the values of all currently defined variables, enter the Set command with no parameters:

```
Set
```

Wildcards (filename generation)

You can specify a number of files at once by using the wildcard characters ? and ≈ (Option-x). The ? character matches any single character (except a colon or Return); ≈ matches any string of zero or more characters (other than colon or Return). For example, the command

```
Files ≈.a
```

lists all filenames in the current directory that end with the suffix ".a". (Several other wildcard characters can also be used to generate filenames—see "Filename Generation" in Chapter 5.)

Locked and read-only files

If you open a file that is locked, or located on a locked disk, the status panel displays a lock icon, as shown in Figure 4-4. When you open a read-only file, the status panel displays a read-only icon, as shown in Figure 4-5. No editing or command execution is allowed in these windows.

■ **Figure 4-4** A locked file with the Lock icon in the Status panel
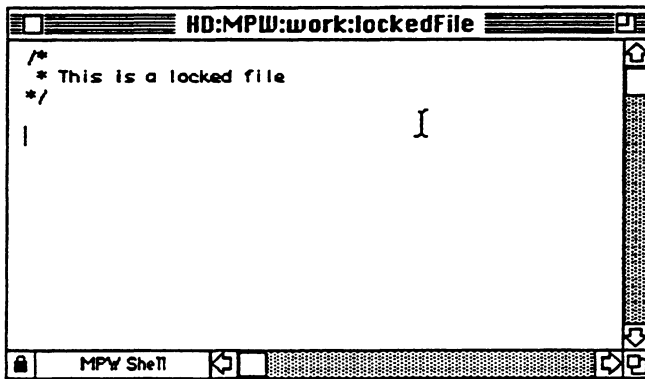
When you check out a read-only copy of a file from a project, this file will always open in read-only mode. The read-only icon is displayed in the status panel, as shown in Figure 4-5.

■ **Figure 4-5**   A read-only file with the Read-Only icon in the Status panel
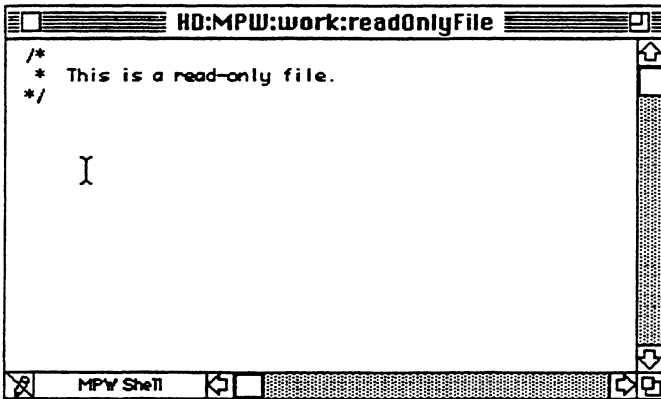
Commando dialogs

The Commando user interface lets you operate any properly configured MPW command by means of a special Macintosh dialog, rather than the traditional command line interface. Commando dialogs may consist of several dialog boxes containing a variety of controls. You can choose options, select filenames, pick directories, and access help information for each option. Commando lets you operate MPW commands in a more intuitive format. All options are visible, and help text for each option can be instantly displayed.

Because of the complexity of many MPW commands, several specialized controls and nested dialog boxes have been implemented for them. The various types of controls and dialog boxes are introduced below. Other dialog boxes, specific to a particular command, appear together with the command in Part II.

Invoking Commando

There are three ways you can invoke a Commando dialog from the Worksheet:

- Option-Enter: Type the command name and then press Option-Enter. This is the easiest method for routine interactive use.

- Ellipsis: Type the command name followed by an ellipsis character (...) and press Enter. You can also use this expression in a script.

  The ellipsis may appear anywhere in a command line (except with quotes or after ∂) and is considered a word-break character. Although the ellipsis may be situated anywhere within the command line, only the first word of the line is actually processed. For example, in the command line

  ```
  addmenu asm alert...
  ```

  only the AddMenu dialog will appear. This results with or without Exit set to 0 or 1.

  The ellipsis invokes the Commando user interface after the Shell has carried out all alias and variable substitutions. The entire command line is passed to Commando and the output of Commando is then executed by the Shell.

  - ◆ *Important Note:* To obtain the ellipsis character, hold down the Option key while simultaneously typing the semicolon (;) character. Although three periods closely resemble an ellipsis character, Commando won't be fooled; you *must* use Option-semicolon to get the true ellipsis character that invokes Commando.

- Type Commando: Type the word Commando in front of the command line and press Enter. This method of invoking Commando only outputs the command line; the command is not executed. You can also use this expression in a script. For example, if you don't want the resulting command line to be immediately executed, you can type

  ```
  commando commandname
  ```

  The tool's frontmost Commando dialog box is displayed. Clicking the Do It button writes the command line to standard output (that is, the window in which you typed the command) instead of executing it immediately. This second method of using dialog boxes is useful for building command lines that are to be cut and pasted into scripts. In this case, Commando will not find a command if the command has been aliased to a different name.

See "Invoking Commando" in Chapter 13 for more information.

Using Commando dialogs

The function and appearance of Commando dialog boxes may vary widely according to the syntax and semantics of the particular command or tool selected. The basic dialog box is typical of a simple command such as Date, the first example used in this chapter. Type

```
Date …
```

Be sure to use Option-semicolon to get the ellipsis. Then press Enter. Figure 4-6 shows the resulting Commando dialog box for Date.

■ **Figure 4-6** The Date dialog box

```
┌─Date Options──────────────────────────────────────────────┐
│  ┌─Date/Time───────────┐ ┌─Amount of Detail──┐ ┌─Date Input──────┐ │
│  │ ◉ Both date and time │ │ ◉ Full date        │ │ Date in Seconds  │ │
│  │ ○ Date only          │ │ ○ Abbreviated date │ │ ┌──────────────┐ │ │
│  │ ○ Time only          │ │ ○ Short date       │ │ │              │ │ │
│  │ ○ In Seconds         │ │                    │ │ └──────────────┘ │ │
│  └─────────────────────┘ └───────────────────┘ └─────────────────┘ │
│              Output                    Error                         │
│           ┌──────────────┐        ┌──────────────┐                  │
│           └──────────────┘        └──────────────┘                  │
├─Command Line──────────────────────────────────────────────┤
│ date                                                        │
├─Help─────────────────────────────────────┬─────────────────┤
│ Show the current date and time.          │ ┌─ Cancel ─────┐ │
│                                          │ └──────────────┘ │
│                                          │ ┌══ Date ══════┐ │
│                                          │ └══════════════┘ │
└──────────────────────────────────────────┴─────────────────┘
```
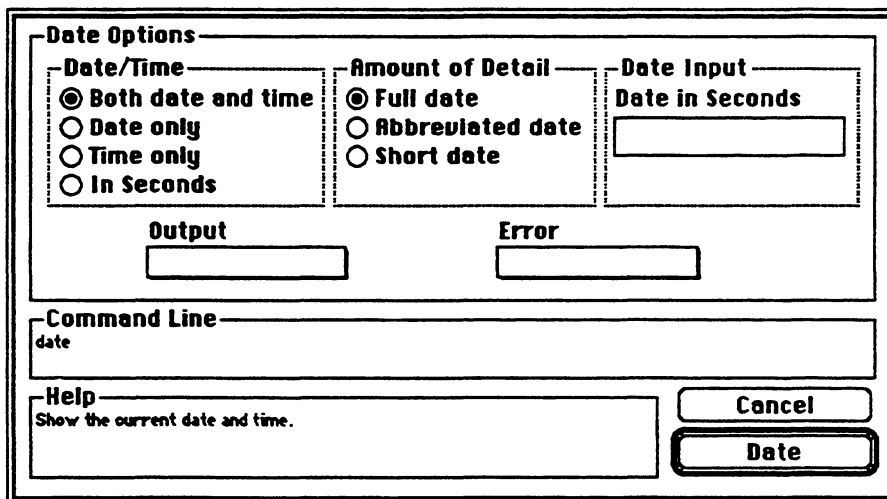
Most dialog boxes share the basic structure shown in Figure 4-6. Various controls for options and parameters appear in the largest, upper area of the box. Date has three parameters:

Date/Time	radio button control
Amount of Detail	radio button control
Date Input	editable field

The default settings for Date appear preselected as the topmost radio button for each parameter.

Clicking and holding down the mouse button on any control or option displays Help information in the standard Help window at the bottom of every Commando dialog box. Clicking on the title of a control also displays the Help information.

Use the pop-up menu of the Output box to redirect output. See the section "Redirecting Output" later in this chapter.

The Command Line window displays the command line resulting from the options you select from the dialog box. As you select options or change parameters, this Command Line box is continuously updated. You can then copy all or any part of the command line using Command-C or the Edit menu.

Clicking the Do It button (the button labelled "Date" in Figure 4-6) passes the completed command line back to the Shell for execution. Alternatively, you can press the Enter key. If you change your mind and decide to exit from the dialog, you can click the Cancel button, which has the same effect as pressing Command–Period.

You can get these special results by holding down the Option and/or Command keys while clicking a Do It button:

Option key (or pressing Enter)	The command line is also written to standard error. This means that the command is executed *and* is echoed to the active window.
Command key	The command line is not passed to the Shell; that is, nothing is executed.
Option key *and* Command key	The command line is written to the active window without being executed.

Standard dialog box controls

This section describes the most frequently encountered Commando dialog box controls.

Generic text parameters

Not only do tools have options, they also have parameters. Nonspecific parameters, where the parameter can be just about any string, are simply entered in an editable text field. For items where text is required, the text is quoted by Commando before being passed to the Shell. You can scroll the line right and left (by dragging) if the text in the box is longer than the text box. Here's an example of an editable text field:

Mark the selection with what name?

```
[                                    ]
```

Repeatable options

Various text field options, such as the -d[efine] option in Rez and Asm, may be specified more than once. The control below shows an option of this type. The number of lines displayed is controllable by the tool's builder. The small window is basically an area where text can be entered, very much like the Notepad desk accessory. This window does not automatically wrap around lines larger than the window area. Instead, it scrolls left and right. You create a new line by pressing the Return key. Scroll the window horizontally by dragging. You can scroll the window's contents vertically either by dragging or by using the scroll bar control.

Preprocessor defines:

```
Language=english        ⬆
size=height*200
                        ⬇
```

Radio buttons

Some options are mutually exclusive and are therefore available as a set of radio buttons. The default setting of the button corresponds to the default state of the option. Groups of mutually exclusive items are often surrounded with a labeled perimeter:

```
┌─Print Quality──┐
│ ○ High         │
│ ◉ Standard     │
│ ○ Draft        │
└────────────────┘
```

Check boxes

An option, such as the Assembler's **-print** option, may have many simultaneous settings. Options like this are implemented with check boxes (versus on/off radio buttons). Most of the MPW tool's options are Boolean flags. Check boxes are also used for these types of options, and are usually surrounded by a labeled perimeter:

```
┌─ Listing Control ──────────┐
│ ☒ Show macro expansions    │
│ ☒ Allow automatic page ejects │
│ ☒ Show warning messages    │
│ ☒ Show macro call statements │
│ ☒ Show generated object code │
│ ☐ Show up to 255 bytes of data │
│ ☒ Show macro directive lines │
│ ☒ Show header lines        │
│ ☒ Show generated literals  │
│ ☐ Show assembly status     │
└────────────────────────────┘
```

Shadow pop-up menus

Some options require the name of a window, alias, font, or Shell variable. Commando will display a field of this type as a shadowed box:

```
Window  [ HD:OS:Worksheet                    ]
```

When you click inside the shadowed box, a pop-up menu displays all the choices for that particular field (that is, windows, aliases, fonts, or Shell variables). The menu box is aligned around the current selection. The current selection is checked in the menu box. As long as the mouse button is held down, the menu behaves like a standard pull-down menu. If necessary, the pop-up menu will scroll vertically. When the mouse is released within a menu item, that item then appears in the shadowed box.

```
Window  [ ✓HD:OS:Worksheet            ]
          HD:MPW:MyCroft:test.c
          HD:MPW:MyCroft:getopt.c
```

Other pop-up variations

Some options are similar to the pop-up menus above but also allow a little more flexibility. The Menu Name box in AddMenu allows you to type in the name of a new menu or select an existing menu name from a list of names:

```
Menu Name [               🗐]
```

Click the menu icon at the right of the box to display a pop-up menu containing the existing choices:

```
Menu Name [       ] 🍎
                     File
                     Edit
                     Find
                     Window
                     Mark
                     Directory
                     Build
```

Drag down the pop-up menu until the item you want is highlighted and then release the mouse button. The selected item will appear in the text edit box. If you type an item into the text-edit box, any identical item in the pop-up menu will be automatically checked.

Multiple input files

When a tool can handle multiple input files of the same type (C, ASM, Rez, and so on), only a single button is displayed.

    [**Source Files**]

Clicking on the button displays a modified standard file dialog box. Commando adds some functionality to the standard file package (SFGetFile) to let you select multiple files in different directories. Another scrollable list appears under the file list. Use the standard file controls to select files and click the Add button to add the selected file to the scrollable list under the SFGetFile. After doing so the dialog box does not disappear. Instead, the file is added to the lower list. (Alternatively, you can just double-click a filename to add it to the lower list.) You can delete a file from the list by selecting it (in the lower list) and clicking Remove. You can select several files at once by holding down the Command key while you click their filenames. When all desired files have been selected, click Done or Cancel to return to Commando's first dialog box.

A tool may tell Commando that the tool requires files with a particular extension. A radio button lets you display and select any text file (or whatever type of file the tool wants). When you select a folder, the Open button reads "Open." When a file is selected, this same button is labeled "Add." If you select a file that has already been added to the lower list, then that file is selected (and scrolled into view if necessary), and the Remove button undimmed.

```
┌─────────────────────────────────────────────────┐
│   ┌───────────────┐                              │
│   │ ⬡ CExamples   │                              │
│  ┌─────────────────────┐                         │
│  │ ▢ Count.c          ⬆│        ⬡ HD             │
│  │ ▢ Memory.c          │                         │
│  │ ▢ Sample.c          │      ┌──────────┐       │
│  │ ▢ Stubs.c           │      │  Eject   │       │
│  │ ▢ testperf.c        │      └──────────┘       │
│  │                     │      ┌──────────┐       │
│  │                     │      │  Drive   │       │
│  │                     │      └──────────┘       │
│  │                     │                         │
│  │                    ⬇│      ┌──────────┐       │
│  └─────────────────────┘      │   Done   │       │
│                               └──────────┐       │
│  ● Files ending in .c   ○ All text files │       │
│  Source:                      └──────────┘       │
│  ┌─────────────────────┐                         │
│  │ :CExamples:Memory.c⬆│      ┌──────────┐       │
│  │                     │      │   Add    │       │
│  │                    ⬇│      └──────────┘       │
│  └─────────────────────┘      ┌──────────┐       │
│                               │  Remove  │       │
│                               └──────────┘       │
└─────────────────────────────────────────────────┘
```
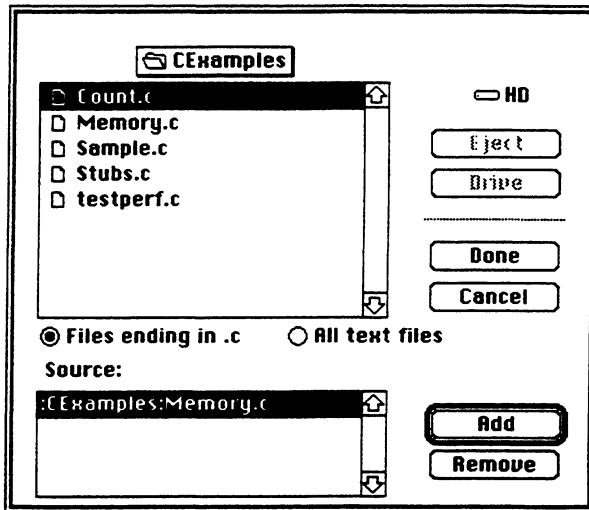
Multiple directories

Some tools, such as C and Asm, have options that let you specify directories to search when looking for various files. Clicking a single button, like this one, will display a modified standard file dialog box:

[Include Directories]

The selection of multiple directories works in the same way as the selection of multiple files. In this example, however, only folders are visible. Because a selected directory has the potential for being both *opened* and *added* to the lower list, there must be two controls for both operations. Clicking the Add button adds the directory selected in the upper list to the lower list. The Open button operates in its normal manner: Clicking it opens the selected folder. You can delete a directory from the lower list by selecting it (in the lower list) and clicking Remove. Finally, clicking Continue or Cancel returns control to Commando.

```
┌─────────────────────────────────────────────────┐
│  ╔═══════════════════════════════════════════╗  │
│  ║  ┌─────────────────────────┐              ║  │
│  ║  │ Add Current Directory:  │              ║  │
│  ║  └─────────────────────────┘              ║  │
│  ║     ┌───────────────────┐                 ║  │
│  ║     │ ⌂ Interfaces      │                 ║  │
│  ║     └───────────────────┘                 ║  │
│  ║  ┌──────────────────────┬─┐               ║  │
│  ║  │ ▢ AIncludes          │⇧│   ⊂▭ HD       ║  │
│  ║  │ ▢ AStructMacs        │ │               ║  │
│  ║  │ ▢ CIncludes          │ │ ┌──────────┐  ║  │
│  ║  │ ▢ PInterfaces        │ │ │  Eject   │  ║  │
│  ║  │ ▣ RIncludes          │ │ └──────────┘  ║  │
│  ║  │                      │ │ ┌──────────┐  ║  │
│  ║  │                      │ │ │  Drive   │  ║  │
│  ║  │                      │ │ └──────────┘  ║  │
│  ║  │                      │ │ ┌──────────┐  ║  │
│  ║  │                      │ │ │  Done    │  ║  │
│  ║  │                      │⇩│ └──────────┘  ║  │
│  ║  └──────────────────────┴─┘ ┌──────────┐  ║  │
│  ║  #Include Search Paths:      │  Cancel  │  ║  │
│  ║  ┌──────────────────────┬─┐ └──────────┘  ║  │
│  ║  │ :Interfaces:RIncludes:│⇧│ ┌──────────┐  ║  │
│  ║  │                      │ │ │  Open    │  ║  │
│  ║  │                      │ │ └──────────┘  ║  │
│  ║  │                      │ │ ┌──────────┐  ║  │
│  ║  │                      │⇩│ │  Add     │  ║  │
│  ║  └──────────────────────┴─┘ └──────────┘  ║  │
│  ║                             ┌──────────┐  ║  │
│  ║                             │ Remove   │  ║  │
│  ║                             └──────────┘  ║  │
│  ╚═══════════════════════════════════════════╝  │
└─────────────────────────────────────────────────┘
```
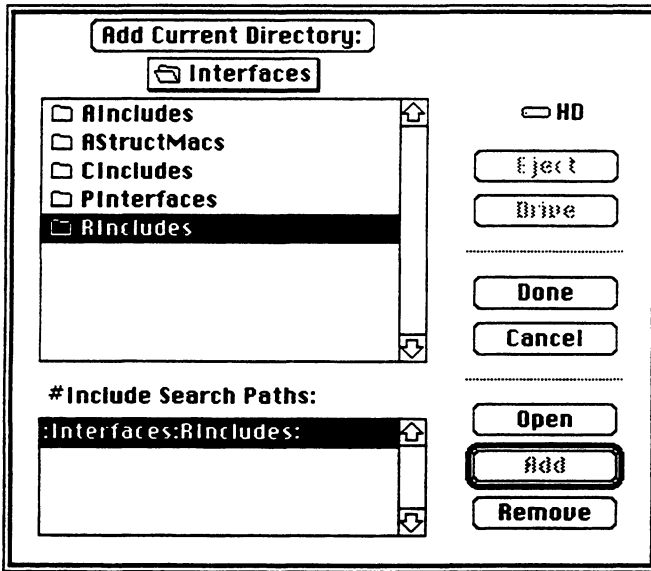
Multiple files and/or directories

For MPW tools or built-in commands that can deal with both multiple files and directories, this dialog box, almost the same as the one shown above, lets you select files and directories. The model is almost the same as the one above, except that both files and folders are visible. Selecting anything in the upper scroll window highlights the lower Add button. The controls work as shown in the example above.

Single input or output file

You select options or parameters that require a single file (whether for input or output) with a control similar to the example below. Clicking in the shadowed rectangle displays a pop-up menu with choices depending upon the tool. The first choice can be either Default Output or No Output (or, if the file is an input file, Default Input or No Input). The Default Output is used for tools that write to a default output file if one is not specified. Link and Rez, for example, write to link.out and rez.out, respectively, if no explicit output file is specified. If Input File or Output File is selected, SFGetFile (for input files), or SFPutFile (for output files) is displayed so that a file can be chosen. If the filename selected is too long to fit in the space provided, the middle of the path is annotated with "...". An ellipsis (typographical; not a Commando invocation) is added to the end of the end if the full filename does not fit within the confines of the box.

```
Resource Output File ┌─────────────────┐
                     │ rez.out         │
                     └─────────────────┘
```

Here's an example of an output file pop-up menu:

```
✓ Write output to Rez.out
  Select an existing output file...
  Write output to a new file...
```
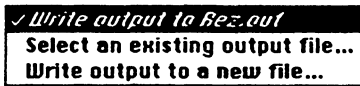
Output file where a file or directory may be specified

The various compilers have options to specify the object filename or the object file directory. Commando displays a pop-up menu similar to this one:

```
File/Directory  ✓ Use default naming convention
                  Specify object file name or select directory...
```

except that the standard dialog box that appears when you select the Output File or Directory item looks like this one:

```
┌─────────────────────────────────────────┐
│  ┌─────────────────────────┐             │
│  │ Select Current Directory: │           │
│    ┌─────────────────┐                   │
│    │ ⬤ AExamples     │                   │
│    └─────────────────┘                   │
│  ┌──────────────────┐⬆   ⬒ HD            │
│  │                  │                     │
│  │                  │    ┌─────────┐      │
│  │                  │    │ Eject   │      │
│  │                  │    └─────────┘      │
│  │                  │    ┌─────────┐      │
│  │                  │    │ Drive   │      │
│  │                  │⬇   └─────────┘      │
│  └──────────────────┘    ┌─────────┐     │
│  Object File/Directory   │Directory│     │
│                          └─────────┘     │
│  ┌──────────────────┐    ╔═════════╗     │
│  │I                 │    ║   OK    ║     │
│  └──────────────────┘    ╚═════════╝     │
│                          ┌─────────┐     │
│                          │ Cancel  │     │
│                          └─────────┘     │
└─────────────────────────────────────────┘
```
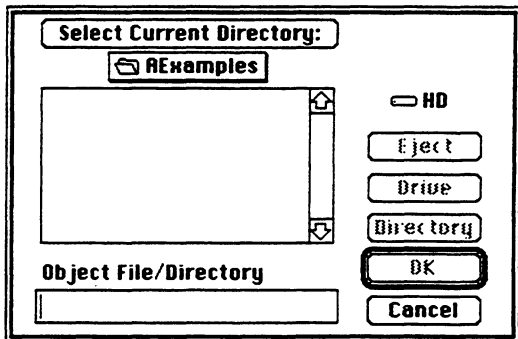
The OK button is dimmed when the text-edit box is empty. After entering text into the text-edit box, the OK button is highlighted. Clicking the OK button specifies the file as the output. Clicking the Directory button specifies a directory as output.

New directories

The NewFolder command lets you specify the creation of multiple directories. The example below (based upon SFPutFile) is used to create multiple directories. When you type a directory name in the middle text-edit area and click the Add button (or press Return), a pathname is added to the lower list. The root of the new directory is the same as that displayed in the upper scroll list. You can continue to add more directories. Click the Done button to close the dialog box and return to the first, or "main" dialog box.
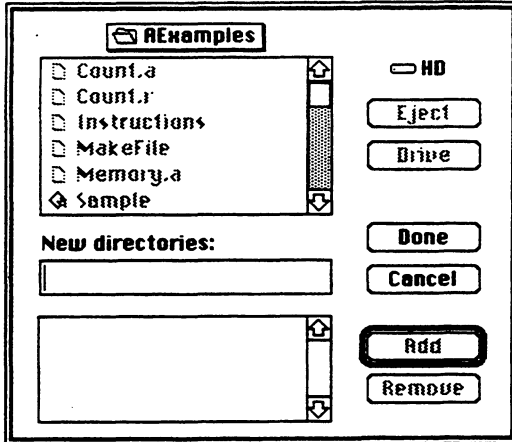
Special dialog box controls

Commando uses standard Macintosh text-edit boxes, radio buttons, and check boxes. In addition to these, you'll encounter some specialized controls because of the variety of options and parameters and certain dependencies between them. These various types of specialized controls are introduced below.

Nested dialog boxes

Some tools, such as Rez and PasMat, have more options and parameters than can fit into one dialog box. The additional options are grouped into nested dialog boxes that are available from the first dialog box. Figure 4-7 below shows, as an example, the first dialog box of Rez.

■ **Figure 4-7**   Rez: the first dialog box
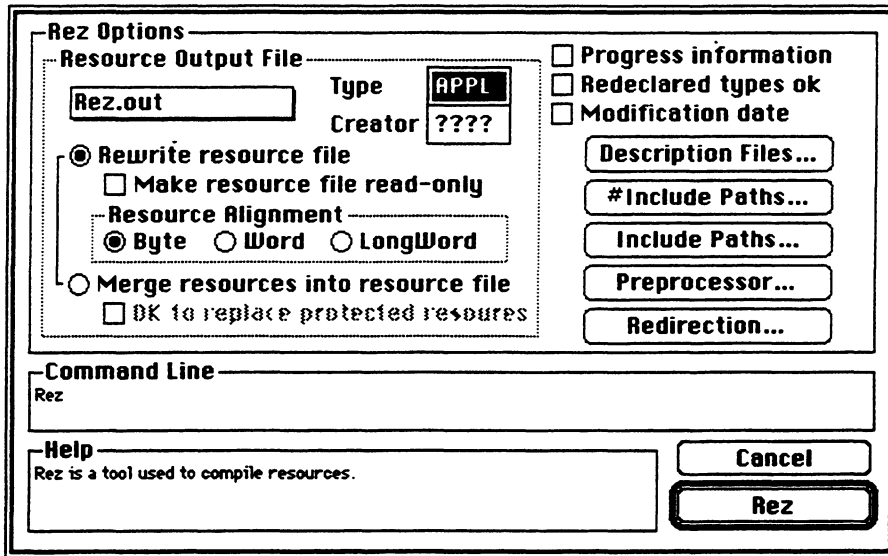
Note the five control buttons at the right side of the "Rez Options" window. When you click one of these buttons, a nested dialog box appears with the title of the selected button. For example, selecting the button labeled "Preprocessor..." displays the nested dialog box shown in Figure 4-8.

■ **Figure 4-8**   Rez: nested preprocessor dialog box
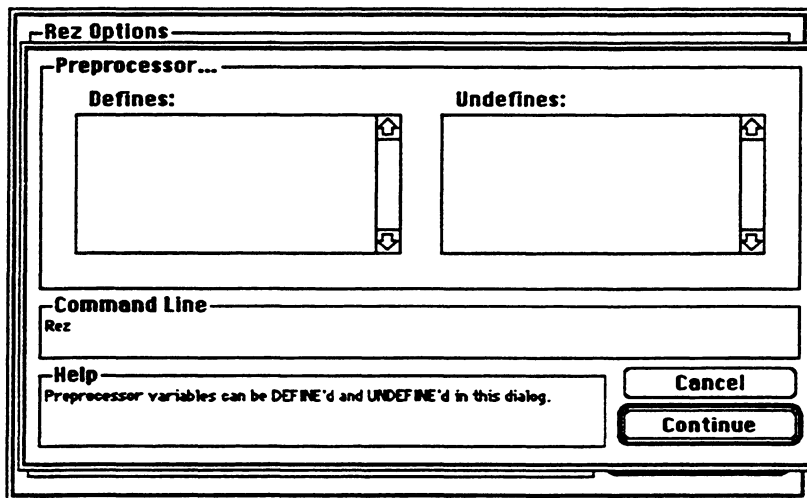
As you type in the preprocessor defines and undefines, the command line you began in the first dialog box is further updated in the Command Line window of the nested dialog box. The lower-right Do It button in a nested dialog box is always labeled "Continue." Clicking Continue closes the nested dialog box, and again displays the first dialog box with the command line updated to show the options and parameters selected in the nested dialog box. (This is always the case, except for the C compiler dialog, which has a third level of dialog boxes.) If you click Cancel, changes from nested dialog boxes are not recorded and you return to the first dialog box. From there you can then select another nested dialog box.

Redirecting output

Every tool that can write information to standard output or to standard error has controls to assign destinations for this output. Consider the Error Output window in the Redirection nested dialog box of Rez, shown in Figure 4-9.

- **Figure 4-9**   Rez: nested Redirection dialog box
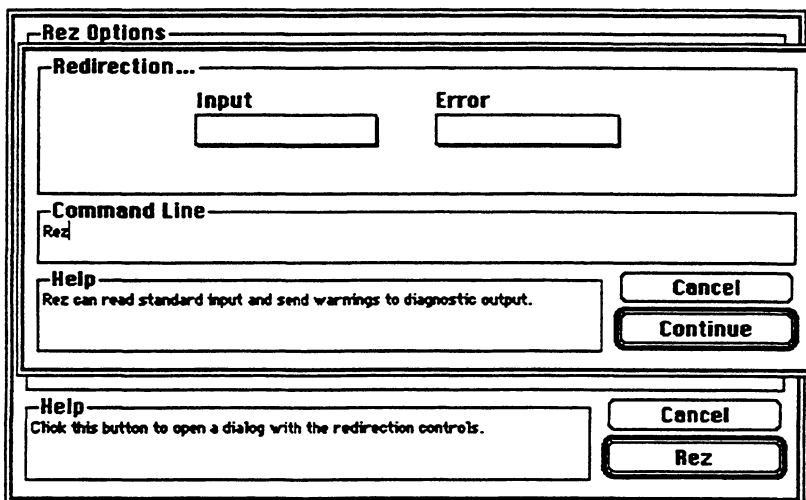
Clicking inside the Error window (and holding the mouse button down) displays this pop-up menu:

```
Error
✓No Output Redirection
 New File...
 Existing File...
 Window...
 Current Selection in Window...
 Current Selection in Target Window
 Standard Output
 Standard Diagnostic
 Null Device                    
 Console
```
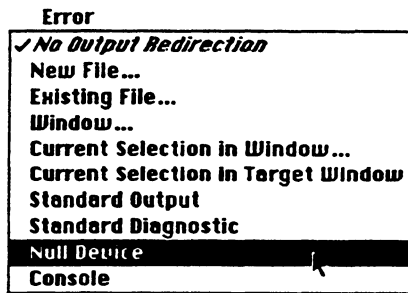
Here Null Device has been selected. When the mouse is released, the filename dev:null appears in the Error window. Whenever you select an output redirection, the two invisible radio buttons directly above the error pop-up are activated.

Selecting "Existing File..." in the pop-up menu displays the standard file dialog box. Selecting "New File" brings up the standard output file dialog box and lets you create a new file. Selecting "Window..." brings up a list of the active windows to choose from. Because a window is a file, you could also choose a window with the Existing File command.

Selecting Current Selection in Window also brings up a list of windows to choose from. When you select Current Selection in Target Window, output is redirected to §. When you choose a window, output is redirected to *window*.§. When you choose any file other than a new file, the Overwrite and Append radio buttons are activated. These buttons correspond to the functions of the $>,\geq$ and $>>$, $\geq\geq$ redirection operators, respectively. Selecting No Output Redirection clears the pop-up menu so that no redirection occurs.

After you release the mouse over Null Device, the command window looks like this:

```
Error   ⊙≥ ○≥≥
Dev:Null          
```

The Diagnostic Output windows and Standard Input windows (in the case of tools that read standard input) work in a similar fashion.

Options dependent on other options

Some options may be dependent on other options. For example, the -hf (header font) and -hs (header size) options of the print tool don't mean anything unless the -h (header) option is specified. Commando implements this model by disabling all controls dependent upon some other control. When you check (or otherwise activate) the main control, the dependent controls are enabled. Another example is the AddMenu command. The syntax of this command is

```
AddMenu [menuName [itemName [command...] ] ]
```

An itemName cannot be entered until a menuName is entered. Likewise, a command cannot be entered until an itemName is entered.

Menu Name [   ] ▤

Item Name [   ] ▤

Command♦
[   ]
[   ]
[   ]

Here is the same set of options after "Find" has been typed in the first text-edit entry field. Notice that as soon as something is entered in the field, the Item Name entry is enabled, but the Commands field remains dimmed.

Menu Name [Find   ] ▤

Item Name [   ] ▤

Command♦
[   ]
[   ]
[   ]

When an item is selected from the Item Name pop-up menu or simply typed into the Item text-edit box, the Commands field is enabled. If Find is a valid menu name, then Find's menu items will appear in the Item Name pop-up menu.

Menu Name [Find   ] ▤

Item Name [Replace   ] ▤

Commands
[⬆]
[   ]
[⬇]

There may be several text-edit boxes that are disabled (dimmed) until you have entered something in the adjacent enabled text-edit box.

Three-state controls

Some options, like the -a option of Setfile, need the support of a three-state control. For example, Setfile can set, clear, or do nothing to the bundle bit. Clicking this control cycles through its three states. The color of the diamond determines its state:

Gray   Don't touch the flag
White   Clear the flag
Black   Set the flag

┌─**Attributes**─────────┐
│   ◇ **Locked**   │
│   ◇ **Invisible**   │
│   ◆ **Bundle**   │
│   ◈ **System**   │
│   ◈ **Protected**   │
│   ◈ **Open**   │
│   ◈ **Changed**   │
│   ◆ **Inited**   │
│   ◇ **on Desktop**   │
└───────────────────────┘
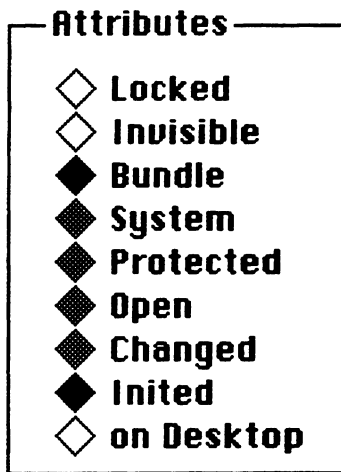
Chapter 5   Using the Command Language

THIS CHAPTER DESCRIBES THE COMPLETE SYNTAX OF THE MPW 3.0 COMMAND
LANGUAGE and explains its use. Each command is defined in detail in Part II. ■

Contents

Overview

The command language provides the following features:

- Built-in and user-definable variables of the form {variableName}
- Command aliases, used to create alternative names for commands
- Command substitution, by which commands enclosed in back-quotation marks (`...`) are replaced by their output
- A quoting mechanism for disabling special characters or inserting invisible characters in text: ∂ literalizes a single character; '...' and "..." quote strings
- An extensive set of structured commands for controlling the order of command execution, including Begin...End, If...Else...End, and For...In...End
- Filename generation with "wildcard" operators such as ≈ and ?
- Redirection of input and output with the <, >, >>, ≥, ≥≥, Σ, and ΣΣ operators

When you enter command text, the Shell first interprets and processes all special symbols before actually running the command. The order of interpretation is explained later in this chapter under "How Commands Are Interpreted." For the most part, the order of presentation in this chapter follows the order of interpretation by the Shell.

In order to begin using MPW, you should read the following sections of this chapter at a minimum:

- The opening sections of the chapter, which describe the basic form of all commands: "Types of Commands," "Entering and Executing Commands," and "Structure of a Command"
- "Command Scripts" and "Special Scripts"
- "Variables"
- "Quoting Special Characters"

The operators and syntax of the command language are summarized in Appendix D.

Types of commands

In all, four kinds of commands are provided:

- Built-in commands, such as Files or Duplicate, are part of the MPW Shell.

- Command scripts, such as Startup, are text files that contain commands. You can combine any series of MPW commands in a text file, and execute the file by entering its filename, just like any other command. You can also pass parameters to a script and use them in commands within the file.

- Tools, such as Link or Asm, are executable programs (that is, separate files on the disk) that are fully integrated with the Shell environment.

- Applications, such as ResEdit or MacPaint®, are stand-alone programs that can be launched from the Shell but can also run outside the Shell environment.

To execute a tool, application, or script, you need to have the proper program file on your disk.

◆ *Note:* A built-in command overrides a script or executable program with the same name. You should therefore use either full pathnames or quotation marks to specify a script or program with the same name as a built-in command. (Quotation marks work for this purpose because the names of built-in commands must appear unquoted—see "Quoting Special Characters" later in this chapter.)

◆ *Note:* The Shell will not execute a tool whose modification date is 12:00 A.M. 1/1/04.

Entering and executing commands

Press the Enter key to execute selected command text. If no text is selected, pressing Enter executes the entire line that contains the insertion point. Alternatively, you can use the mouse to click the Status Panel in the Worksheet's lower–left corner, or press Command-Return; both methods have the same result as pressing the Enter key.

△ **Important**   If no text is selected, pressing Enter always passes the entire line to the Shell (or to whatever other program happens to be reading from the console). This rule also applies to your own integrated programs that run within the Shell. △

△ **Important**   If you enter a line that ends with the Shell escape character, ∂, the command interpreter will pause, waiting for the rest of the line. △

All commands return a **status code:** 0 indicates successful completion; nonzero values usually indicate an error. This code is returned in the {Status} variable, described later in this chapter.

Negative status codes

The command interpreter will return negative status codes when it encounters an error. These codes are:

−1   Command not found, script is a directory, script is not executable, or script has a bad date.

−2   Filename expansion failed, or there was an error in the expression syntax.

−3   Bad syntax. Quotation characters and braces were not balanced, or were missing end or ")" command. Error in control constructs.

−4   Missing filename following I/O redirection or the file could not be opened.

−5   Invalid expression (If, Break If, Continue If, and other such constructs).

−6   Tool could not be started.

−7   Runtime error during tool execution, most likely an out-of-memory error.

−8   User aborted the tool from the debugger.

−9   User aborted the tool with Command-period.

These values can be used to distinguish between errors returned by the commands themselves and errors returned by the Shell.

△ **Important**   All negative numbers are reserved for the Shell. Use only positive numbers for errors in tools or scripts. △

Structure of a command

A command is written as a list of words separated by blanks. (Blanks may be either space or tab characters.) The first word is the name of the command, and each word that follows is passed as a parameter to the command. The general form of a simple command is

commandName [*parameters...*] *commandTerminator*

Each of these elements is described below.

Command name

The **command name** is either the name of a built-in command or the filename of the program or script to execute. Command names are not case sensitive. Alternative names can be defined for a command—see "Command Aliases" in this chapter for information.

The command name is passed to tools and scripts as parameter 0, and can be referenced by scripts in the variable { 0 }, explained later in this chapter under "Variables."

Parameters

Each of the subsequent words in a command is a **parameter** to the command or to the command interpreter. Note that certain parameters, such as I/O redirection, are interpreted by the Shell, and never seen by the command itself. Variables are also interpreted before being passed to the program.

By convention, there are two distinct types of parameters to commands: **options** and **files.** See the "Command Prototype" section at the beginning of Part II for more details on these conventions.

You can reference parameters within scripts by using the variables {1}, {2},...{n}. (See Table 5-5.)

Command terminators

Each command is normally terminated by a return character. Commands can also be terminated by the pipe symbol (|), the conditional execution operators (&& and | |), or the simple command terminator (;). Each of these symbols may be followed by a return. Table 5-1 describes the command terminators in order of decreasing precedence.

Except as modified by structured commands, commands are read sequentially and executed as they are read.

■ **Table 5-1**   Command terminators

Command	Description
cmd1	cmd2	Saves the standard output of cmd1 in a temporary file and uses it as the standard input of cmd2. (Standard I/O is explained later in this chapter.)
	Note: In MPW, unlike UNIX® systems, the commands are executed sequentially.
cmd1 && cmd2	Executes cmd2 only if cmd1 succeeds (that is, returns a status value of zero).
cmd1		cmd2	Executes cmd2 only if cmd1 fails (returns a nonzero status value).
cmd1 ; cmd2	Executes cmd1 followed by cmd2; this terminator allows more than one command to appear on a single line.

These command terminators may be applied to both simple and structured commands. Grouping is from left to right. You can use parentheses to group commands for conditional execution and pipe specifications. Here are some examples:

```
Files | Count -l
```

This command pipes the output of the Files command (a list of files and directories) to the Count command, which counts the lines in the list.

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

This example begins by assembling Sample.a. If that operation succeeds, it links the object file; but if the assemble-and-link operation fails, it echoes the message "Failed," and beeps.

Command continuation

You can continue a command onto the next line by typing ∂ (Option-D) followed by a return. Both characters are discarded when the line is interpreted. The return must come *immediately* after the ∂, with no blanks or comments between them. (For more information about the ∂ escape character, see "Quoting Special Characters" in this chapter.)

```
Echo  This is all ∂
one command

This is all one command
```

Notice that the output appears on one line.

Comments

The number sign (#) indicates a comment. Everything from the # to the end of the line is ignored. (Comments *always* end at the next return, even if the return is preceded by a ∂.)

```
Echo  This is echoed.        # This is not.
Echo parameters              # comment  ∂
      more parameters        # another comment
```

Simple versus structured commands

All of the commands introduced so far have been **simple commands.** Simple commands consist of a single keyword, followed by zero or more parameters. Simple commands are distinguished from **structured commands**—commands such as For and If, for example, that let you control the order in which other commands are executed. For example,

```
For file In ≈.c; Count {file}; End
```

All structured commands are built-in, and usually have more than one keyword. The entire structured command is read before its execution begins.

Also see "Structured Commands" in this chapter.

Running an application outside the Shell environment

You can run an application outside the MPW Shell environment by executing the program name just like any other command. For example,

```
ResEdit
```

The application is loaded and launched as if it had been started from the Finder. Any files specified as parameters are passed to the program via the application parameter handle, in Finder fashion. (See "Finder Information" in the chapter "Segment Loader" of *Inside Macintosh*.) The following option is available on the command line:

-p *file...*   Tell the program to print the specified files.

For example,

```
MacPaint -p "HD:Screen 1" "HD:Screen 2"
```

This command tells the Shell to run MacPaint (assuming MacPaint is in a directory listed in the Shell variable {Commands}), and to print the files Screen 1 and Screen 2.

The Shell environment is saved when the application is launched and restored when the application terminates. (These actions are performed by the Suspend and Resume command files, described below.)

◆ *Note:* When running MPW under MultiFinder, the application is launched into a separate MultiFinder partition and the state is not saved.

▲ **Warning**   Running an application from a script normally terminates the script. Under MultiFinder, the application starts and the script continues to execute. ▲

Scripts

You can create your own commands by writing text files of previously defined commands, called **scripts** (command files). You can execute such a file just like any other command within the Shell environment—the name of the file you created is the name of the new command. For example,

```
Date
Echo Volumes..........................................
Volumes
Echo Current Directory................................
Directory
Echo Files............................................
Files
```

If this text is on the screen, you can execute it by selecting it and pressing Enter. You can also save this text as a script so that it's always available. To save it under the name "Info," for example, first select the command text, making sure that the window with the selected text is the target (second from the front) window. Then type the following command in another window:

```
Duplicate -d § Info
```

You can now execute this series of commands by entering the command name Info. (Recall that the § character indicates the selection in the target window.)

You can pass parameters to a script just as you would to a predefined command by using the normal Shell syntax:

filename [*parameters...*]

Parameters can be referred to within the scripts by using the built-in variables {1},
{2},... {n}, explained below under "Parameters to Scripts."

◆ *Note:* As a matter of convenience, scripts (as well as applications and tools) are usually kept in directories that the Shell automatically searches when a leafname is given for a command name. This convention allows you to invoke the command by using its leafname instead of its full pathname. The Shell variable {Commands} contains a comma-separated list of directories to be searched. You can easily modify it to include additional directories.

Special scripts

The scripts described in this section are provided with MPW. You can modify the commands in each of these files to suit your needs.

△ **Important**   Each of these scripts must be in the same directory as the MPW Shell, or in the System Folder. △

The Startup and UserStartup files

When you start up the Shell, commands are initially read from a file named Startup. The Shell executes the commands in Startup as if you had entered them interactively. The Startup file provided with MPW contains several default variable and alias definitions. You can modify the commands in Startup to suit your own needs; for instance, you can change the default pathnames to suit a special directory configuration.

Startup executes another script called UserStartup. It's recommended that you use this file for your own changes and additions to the startup sequence. You can redefine the variables defined in Startup, set and export any number of additional command-language variables, and define aliases and create menu items. Aliases and variables are fully described in the sections that follow.

Suspend, Resume, and Quit

When you run an application from the Shell, commands are read from the file Suspend. When you quit the application and return to the Shell, commands are read from the file Resume. The Suspend and Resume files save state information about variable definitions, exports, aliases, and windows before running an application; they then restore the state after returning to the Shell.

◆ *Note:* Suspend and Resume are not used if the MPW Shell is running under MultiFinder.

When you quit from the Shell, commands are read from the file Quit. The Shell executes these commands before closing any windows.

◆ *Note:* If you cancel from the Quit command, the Quit file will already have been executed.

Like Startup and UserStartup, these scripts run as if you had entered the commands interactively. You can modify them to suit any special requirements you may have.

Command aliases

An **alias** is an alternative name for a command (and possibly some parameters). The Alias command is used to define aliases and to display the list of aliases. If an alias has been defined, it will be recognized by the command interpreter and the corresponding definition will be substituted.

◆ *Note:* Variable substitution and alias substitution occur on the alias definition itself after it has been substituted.

The following commands are used to define and undefine aliases:

Alias *name word...*   *Name* becomes an alias for the list of words.
Alias *name*   Displays any alias definition associated with *name*.
Alias   Displays all alias definitions.
Unalias *name*   Removes any alias definition associated with *name*.
Unalias   Removes all alias definitions.

Aliases are local to the script in which they are defined (and are globally available if they are defined in the Startup and UserStartup files or entered interactively). Aliases are automatically inherited from enclosing scripts, and they may be redefined locally. However, aliases redefined locally will revert to their previous value when the script terminates.

See the Alias and Unalias commands in Part II for a complete specification of aliases and several examples.

Executable error messages

The following alias is defined in the Startup file:

```
Alias File Target
```

That is, the word "File" is defined as an alias for the Target command, which opens a file as the target window. (See Chapter 6, "Editing Commands.") This alias is useful when a compiler returns an error message such as

```
### Not a parameter name:  counts
    File "Count.c" ; line 73
```

By placing the insertion point anywhere on the error message line or by selecting the entire line and pressing the Enter key, you'll automatically open the specified file as the target window, find and select the offending line, and bring the window to the top. The command that the Shell actually executes is

```
Target "Count.c" ; Line 73
```

Line is a script that automatically finds and selects a line by number and then brings the target window to the top.

Variables

The Shell provides several predefined variables and allows you to declare any number of additional variables. Variables are used for

- shorthand notation
- status information
- local variables in scripts
- parameters to scripts and tools
- certain defaults for the MPW Shell

You can define or redefine variables with the Set command and remove variable definitions with the Unset command. For example, the command

```
Set PFiles HD:MPW:PFiles:
```

defines a variable {PFiles} with the value "HD:MPW:PFiles:".

Variables have strings as their values. You can reference them by using the notation {name}, where name is the name of the variable. When a command containing a variable {name} is executed, {name} is replaced with the current value of the variable. In this example,

```
Files {PFiles}Src.p
```

{PFiles} is replaced with its definition before the command is executed.

A variable may comprise one or more words, or part of a word. If a variable is undefined, {name} is removed (that is, replaced with a string of length zero, called a **null string**).

Variable names are case insensitive, and must not include the right brace character (}), for obvious reasons. It's wise to avoid using any special characters in variable names because future extensions to the command language may assign special meanings to some of these characters.

◆ *Note:* For variables such as {Exit} and {CaseSensitive} that can be either "true" or "false," the variable is considered true if it is set to anything other than zero or the null string (a string of length zero). The variable is considered false if it is set to zero, null, or undefined. The best way to set one of these variables is like this:

Set Exit 1	# turn {exit} on
Set Exit 0	# turn {exit} off

(These values also apply to expressions that return a Boolean value, defined later in this chapter under "Structured Commands.")

Predefined variables

Table 5-2 lists the variables defined by the MPW Shell. These variables provide the status value returned by the last command as well as the pathnames of several files and directories.

△ **Important**   Since the variables listed in Table 5-2 are predefined or defined dynamically by the Shell, you should not modify the values of these variables. △

■ **Table 5-2**   Variables defined by the Shell

Variable	Description
{Active}	Full pathname of the current active window.
{Aliases}	A list of all defined aliases, with each name separated by a comma. The list contains only the names, not the definitions. Commando uses this variable with the built-in commands Alias and Unalias. Commando needs this variable to know the names of existing variables. {Aliases} must be exported.
{Boot}	Volume name of the boot disk.
{Command}	Full pathname of the last command executed. (For built-in commands, this is the name of the command.)
{ShellDirectory}	Full pathname of the directory that contains the MPW Shell.
{Status}	Result of the last command executed. (A value of 0 means successful completion. Any other value is an error code: Typically, 1 means an error in parameters, and 2 means that the command failed.)
{SystemFolder}	Full pathname of the directory that contains the System and Finder files.
{Target}	Full pathname of the target window. The target window is the second window from the top; by default, this is the window where editing commands (such as cut, copy, and paste) take effect.
{Windows}	Contains a list of the current windows, with each name separated by a comma. Commando uses this list to allow redirection of output or input to or from existing windows. Commando needs this variable to know the names of the current windows. {Windows} must be exported.
{Worksheet}	Full pathname of the Worksheet window.

Variables defined in the Startup file

Table 5-3 lists the variables defined in the Startup file (described in the "Special Scripts" section earlier in this chapter). These variables define pathnames and default settings to the Shell and are referenced by the Shell and by some of the MPW tools. You can change any of these definitions to suit your preferences.

Hierarchical file system (HFS) pathname conventions are described in Chapter 4.

■ **Table 5-3**   Variables defined in the Startup file

Variable	Description

Variables referenced by the command interpreter

{Commando}   This variable tells the Shell which command to execute when the ellipsis character (Option-semicolon) is present anywhere in a command line. The Startup file sets this variable to "Commando." The {Commando} variable allows the development of similar tools whose output is to be executed by the Shell. If the variable is not set, then the ellipsis character is removed from the command line and normal execution proceeds. {Commando} must be exported if scripts are to use Commando.

{MPW}   The volume or folder containing the Macintosh Programmer's Workshop. Initially set to the directory containing the MPW Shell. If you put the MPW Shell on your desktop, modify the value of {MPW} in the Startup file.

{Commands}   A list of the directories that the Shell searches when looking for a command to execute. Directories in the list are separated by commas. A single colon indicates the default directory. {Commands} is initially set to

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

—that is, the current directory; then HD:MPW:Tools, then HD:MPW:Scripts, and then HD:MPW:Applications (assuming that {MPW} is set to HD:MPW:).

{Echo}   When {Echo} is set to a nonzero value, commands are written to diagnostic output after aliasing, variable substitution, command substitution, and filename generation, and just prior to execution. This capability is useful for watching the progress of a script and for debugging scripts. As the first line of your file, include the line

Set Echo 1

{Echo} is initially set to 0.

{Exit}   When {Exit} is set to a nonzero value, scripts terminate whenever a command returns a nonzero status. This nonzero status is returned as the status value of the script. (See the {Status} variable in Table 5-2.) {Exit} is initially set to 1.

{Test}   When {Test} is set to a nonzero value, the command interpreter executes built-in commands and scripts, but not tools or applications. {Test} is useful for checking the control flow in command files. (It's most useful if {Echo} is also nonzero.) {Test} is initially set to 0.

(Continued)

■ **Table 5-3** (Continued) Variables defined in the Startup file

Variable	Description

Variables referenced by the editor

{AutoIndent}   Specifies the setting for automatic indenting. The default setting for a new window is 1. If {AutoIndent} is set to any value greater than 0, automatic indenting occurs.

{CaseSensitive}   Any nonzero value specifies case-sensitive pattern matching. {CaseSensitive} is initially set to 0 (that is, false). You can also set {CaseSensitive} from the Find and Replace dialog boxes, by clicking the Case Sensitive button. (See "Find Menu" in Chapter 3.)

{Font}   Specifies the font for a new window. Its predefined value is "Monaco."

{FontSize}   Specifies the font size for a new window. It is preset to 9.

{PrintOptions}   Options used by the Print Window and Print Selection menu items. They are initially set to -h. (The -h option prints pages with headers. For more information on possible print options, see the Print command in Part II.)

{SearchBackward}   If set to any nonzero value, searching will proceed backward. This variable can be used to set up the default environment so that you can access the backward search option. The default value is 0. You can also set {SearchBackward} from the Find and Replace dialog boxes by clicking the Search Backward button. (See "Find Menu" in Chapter 3.)

{SearchType}   Use this variable to set up the default environment so you can access selective search options. If {SearchType} is set to 0, the search will find the literal character string specified. If it is set to 1, only words will be searched. If set to 2, regular expressions will be searched. The default value is 0. You can also set {SearchType} from the Find and Replace dialog boxes by clicking one of the Literal, Word, or Selection Expression buttons. (See "Find Menu" in Chapter 3.)

{SearchWrap}   Use this variable to set up the default environment for wrap-around searching. If set to any nonzero value, searching will wrap around. The default value is 0. You can also set {SearchWrap} from the Find and Replace dialog boxes, by clicking the Wrap Around button. (See "Find Menu" in Chapter 3.)

{Tab}   Default tab setting for new windows (initially 4).

(Continued)

■ **Table 5-3** (Continued) Variables defined in the Startup file

Variable	Description

Variables referenced by the editor (Continued)

{User}
The name of the current user of MPW, predefined to be the same as the user name specified in the Chooser.

{WordSet}
The set of characters that constitute a word to the editor (for use with Find and Replace menu commands, and for word selection by double-clicking). By default, {WordSet} is set to the characters a–z, A–Z, 0–9, and _ (underscore). If a character is not in the list, the editing commands regard it, like a blank, as a break between words.

Pathnames for libraries and Include files

{AIncludes}
The directories to search for assembly-language Include files, referenced by the Assembler. Initially set to

"{MPW}Interfaces:AIncludes:".

{CIncludes}
The directories to search for C Include files, referenced by the C compiler. Initially set to

"{MPW}Interfaces:CIncludes:".

{CLibraries}
The directory that contains C library files. Initially set to

"{MPW}Libraries:CLibraries:".

{Libraries}
The directory that contains shared library files. Initially set to

"{MPW}Libraries:Libraries:".

{PInterfaces}
The directories to search for Pascal interface files, referenced by the Pascal compiler. Initially set to.

"{MPW}Interfaces:PInterfaces:"

{PLibraries}
The directory that contains Pascal library files. Initially set to

"{MPW}Interfaces:PLibraries:".

{RIncludes}
The directory that contains Resource compiler (Rez) Include files. Initially set to

"{MPW}Interfaces:RIncludes:".

UserVariables

UserVariables is a script that lets you use Commando to create Set commands for user variables that you may wish to include in your startup script. Paste the command line created by Commando into your User Startup file and format it as you like. Note that the commands are separated by semicolons. Don't forget to remove the UserVariables command from the beginning of the command line.

The variables in the UserVariables script are divided into six groups:

Control Variables	{Echo}, {Exit}, {IgnoreCmdPeriod}, and {Test}
Search Variables	{SearchType}, {CaseSensitive}, {SearchBackward}, {SearchWrap}, and {WordSet}
Print Options	{PrintOptions}
Window Stacking	{StackOptions}
Window Tiling	{TileOptions}
Window Variables	{NewWindowRect}, {ZoomWindowRect}, {AutoIndent}, {Font}, {FontSize}, and {Tab}

These variables are described in Table 5-4 that follows and in Table 5-3 in the previous section.

■ **Table 5-4**   User variables not defined in Startup file

Variable	Description
{DirectoryPath}	Use this variable to change directories easily. {DirectoryPath} is searched by the Directory command when you attempt to set a directory by using only its leafname. (See Directory in Part II.)
{IgnoreCmdPeriod}	This variable tells scripts to ignore Command-Period. This is useful for critical sections of a script. If this variable is set to a nonzero number, Command-Period is ignored. Tools that run in the scope that has {IgnoreCmdPeriod} defined will also ignore Command-Period. This overrides any signal handler defined in the tool itself. {IgnoreCmdPeriod} is undefined at startup.

△ **Important**   If {IgnoreCmdPeriod} is set, the only way to prematurely stop execution is to reboot. △

{NewWindowRect}	Specifies the window size when a new window is created. The value of this variable is the four coordinates of a rectangle, listed in this order: top, left, bottom, right. The defined rectangle must be visible on the Macintosh screen. If the rectangle specified is not totally visible it is clipped to the edges of the screen. The coordiates (0,0) are at the left side of the screen at the bottom of the menu bar. For example, to create all new windows in the top left corner of the screen 400 pixels wide and 200 pixels high, use the following command:

```
Set NewWindowRect 0,0,400,200
```

{StackOptions}	Options used by the Stack Windows menu command. Use this variable to specify your own preferences. (See "Window Menu" in Chapter 3.)
{TileOptions}	Options used by the Tile Windows menu command. Use this variable to specify your preferences. (See "Window Menu" in Chapter 3.)
{ZoomWindowRect}	
	Specifies the size of a window when it is zoomed to full screen size. The value of this variable is the four coordinates of a rectangle, listed in this order: top, left, bottom, right. The defined rectangle must be visible on the Macintosh screen. If the rectangle specified is not totally visible, it is clipped to the edges of the screen. The coordinates (0,0) are at the left side of the screen at the bottom of the menu bar.

Parameters to scripts

When a script is executed, the values of certain Shell variables are set automatically. These variables are explained in Table 5-5.

■ Table 5-5 Parameters to scripts

Variable	Description
{0}	Name of the currently executing script.
{1}, {2},...{*n*}	First, second, and *n*th parameter passed to the current script. (These values are null for commands entered interactively.)
{#}	Number of parameters (excluding the command name).
{Parameters}	Equivalent to {1} {2} ...{*n*}.
{"Parameters"}	Equivalent to "{1}" "{2}" ..."{*n*}". This form should be used if the parameters could contain blanks or other special characters.

The {Parameters} variable is especially useful when the number of parameters is unknown. The quoted forms, such as "{1}" or {"Parameters"}, are usually preferable to the unquoted forms because, after variable substitution, {1}, {2}, and so on could contain blanks or other special characters. For example, consider the Line script (which is useful with error messages, as explained earlier in this chapter under "Executable Error Messages"):

```
Find "{1}" "{Target}"    # Find line n in the target window.
Open "{Target}"          # Make the target window the active  ∂
                         # (top) window.
```

This script takes one parameter, a line number. Parameter {1} is quoted to handle the case where Line is called without any parameters. In this case the value of {1} is the null string, and without the quotes the {1} would completely disappear, leaving the name of the target window as the only parameter to Find. The quotation marks ensure that at least a null string is sent to Find as its first parameter—this is essential, because the window name must be the second parameter. Also notice that the {Target} variable is quoted, because it is a filename that might contain blanks or other special characters. (For more information on quoting rules, see "Quoting Special Characters" later in this chapter.)

Defining and redefining variables

The following commands are used to define and modify variables:

Set *name value*   Assigns the string *value* to variable *name*.

Set *name*   Writes the value of variable *name* to standard output.

Set   Writes a list of all variables and their values to standard output.

Unset *name*   Removes the definition of variable *name*.

Unset   Removes the definition of *all* variables in the current scope. (For an explanation of the scope of a variable, see the next section.)

▲ **Warning**   Removing all variables in the outermost scope can have serious consequences. For example, the Shell uses the variable {Commands} to locate MPW tools and other commands. The assembler and compilers use other variables to help locate Include files. Some variables, such as {Boot}, cannot be reinitialized without restarting MPW. ▲

Defining a variable and making it available for use by scripts and programs involves two separate steps:

1. You can define a variable with the Set command. Note that variables are local to the script in which they are defined—a variable definition ceases to exist when its command file terminates.

2. You can pass a variable to scripts and tools with the Export command. After you export a variable, nested scripts can reference that variable and may override its value locally—but any redefinition is strictly local and terminates when the script terminates. It's impossible to affect the value of a variable in an enclosing script. (See Figure 5-1.)

Exporting variables

The Export command makes variables available to scripts and tools:

Export *name*...   Exports the named variables.
Export   Writes the list of exported variables to standard output.
Unexport *name*...   Removes specified variables from the list of exported variables.
Unexport   Writes the list of unexported variables to standard output.

You can define a variable globally by setting its value in the Startup file and exporting it. Figure 5-1 illustrates how Export works.

■ **Figure 5-1**   Trafficking in variables

```
### UserStartup File ###
Set var X
Export var

# (var) = 'X'
ACommandFile

        ### ACommandFile ###
        Set Var Y
        Export var
        Set local Z
        AnotherCommandFile

                ### AnotherCommandFile ###
                # (local) is undefined
                # (var) = 'Y'
                Set var Z
                # (var) = 'Z'

        # (var) = 'Y'
        # (local) = 'Z'

# (var) = 'X'
```
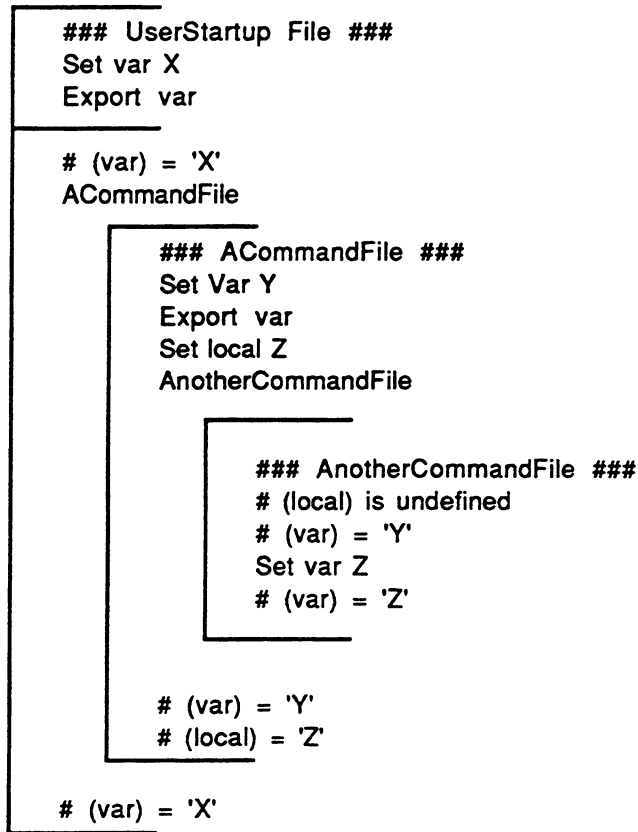
◆ *Note:* You can use the Execute command to execute a script without creating a new scope for variables, exports, and aliases. The Shell "executes" the Startup, Suspend, Resume, and Quit scripts, and Startup uses Execute to run the UserStartup script. For more details about Execute, see Part II.

Command substitution

Command substitution causes a command to be replaced by its output. You can specify command substitution by enclosing one or more commands in backquotes (`...`). The backquote key is located at the upper-left corner of the original Macintosh keyboard; it is located near the space bar of the newer keyboards. When the command is executed, the standard output of the enclosed commands replaces the `...`. Command substitution can form part of a word, a complete word, or several words. Command substitution is not done within "hard" quotation marks (that is, the standard single quotation marks ' ... ').

◆ *Note:* If the standard output of the enclosed commands contains return characters, the returns are replaced by blanks. If the output ends with a return, this return is discarded.

For example, the command

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

echoes the parameters, replacing the Date command with its output, as follows:

```
The date is Wednesday, October 22, 1987 10:40:00 PM
```

The following example duplicates the files whose names are output by the Files command:

```
Duplicate `Files -t MPST MyDisk:` "{MPW}Tools"
```

The command line

```
`Files -t MPST MyDisk:`
```

is replaced with a string of filenames of type MPST (that is, MPW tools) before the Duplicate command is executed; these files are then copied to the folder {MPW}Tools. This command is useful because the Files command allows you to specify files with a certain type or creator, something you can't do with wildcard operators.

Filename generation

After variables have been substituted, an unquoted word that contains any of the characters

```
?    ≈    [    *    +    «
```

is considered a filename pattern. The word is replaced with an alphabetically sorted list of filenames that match the pattern. An error is returned if no filename is found that matches the pattern.

You can specify a group of file- (or window-) names with the "wildcard" notation given in Table 5-6.

■ **Table 5-6**   Filename generation operators

Variable	Description
?	Matches any single character (except a colon).
≈	Matches any string of zero or more characters (except a colon).
[*characterList*]	Matches any character in the list.
[¬ *characterList*]	Matches any character not in the list.
*	0 or more repetitions of the preceding character or character list (?* is the same as ≈).
+	1 or more repetitions of the preceding character or character list.
«number of repetitions»	Specifies number of repetitions of the preceding character or character list.

The pathname separator (:) must appear explicitly in the pattern because the : character will never be substituted for ?, ≈, or [...].

◆ *Note:* Pattern matching is not case sensitive.

These special characters are the same **regular expression operators** used in editing commands. For a complete discussion of regular expressions, see Chapter 6.

Naturally, you need to be careful with these wildcard operators. The Parameters and Echo commands are very useful for double-checking which filenames a command will generate. For example, before giving the command

```
Delete ≈.c.o
```

you might want to run the command

```
Parameters ≈.c.o
```

This command lists your ".c.o" files to standard output so that you can make sure you really want to delete them all.

◆ *Note:* Wildcard characters only generate names that match existing filenames; they do not create new files. For example, the following attempt to rename files *will not work:*

```
Rename ≈.obj ≈.o
```

An example of how to perform a wildcard rename can be found under the description of the Rename command in Part II.

Quoting special characters

There are numerous characters that have special meanings to the MPW Shell. Normally, the Shell performs the action indicated by the special character—but you can disable a character's special meaning (that is, include it as a literal character) by quoting it. You commonly need quotes when specifying filenames that contain blanks or other special characters or when searching for the literal occurrence of a special character. See also "Pattern Matching" in Chapter 6.

Table 5-7 lists all of the special symbols recognized by the Shell.

■ **Table 5-7**   Special characters and words

Character	Meaning	Where described
Space	Separates words	"Structure of a Command"
Tab	Separates words	
Return	Separates commands	"Structure of a Command"
;	Separates commands	Table 5-1
|	Separates commands, piping output to input	
&&	Separates commands, executing the second if the first succeeds	
| |	Separates commands, executing the second if the first fails	
(...)	Command grouping; grouping in filename generation	
...	Invokes Commando	"Invoking Commando"

♦ *Note:* This ellipsis character is an Option-semicolon key command, *not* three periods.

Character	Meaning	Where described
#	Comments	"Structure of a Command"
∂	Escape character: quotes the subsequent character	In this section (Table 5-8)
'...'	Quotes all special characters	
"..."	Quotes all special characters, except ∂, {, and `	
/.../	Quotes all special characters, except ∂, {, and `	
\...\	Quotes all special characters, except ∂, {, and `	
{...}	Variable substitution	"Variables"
`...`	Command substitution	"Command Substitution"
?	Matches a single character in filename generation.	"Filename Generation" In this chapter
≈	Matches any string in filename generation	"Pattern Matching" in Chapter 6
[...]	Character list in filename generation	
*	Zero or more repetitions in filename generation	
+	One or more repetitions in filename generation	
« »	Specified number of repetitions in filename generation	

(Continued)

■ **Table 5-7** (Continued) Special characters and words

Character	Meaning	Where described
<	Input file specification	"Redirecting Input and Output"
>	Output file specification	Table 5-12
>>	Output file specification (append)	
≥	Diagnostic file specification	
≥≥	Diagnostic file specification (append)	
Σ	Output file and diagnostic file specification	
ΣΣ	Output file and diagnostic file specification (append)	

You can literalize a character by preceding it with the Shell escape character, ∂ (Option-D), or by including it within the quotation symbols '...', "...", /.../, or \...\. The escape character, ∂, quotes a single character only; the other quotation symbols may be used to quote part or all of a word. These symbols are described in Table 5-8.

■ **Table 5-8** Quotes

Quote	Description
'...'	"Hard quotation marks": Take the enclosed string literally—no substitutions occur. The quotation marks are removed before execution.
"..."	"Soft quotation marks": Take the enclosed string literally. ∂c, variable substitutions, and command substitutions occur. The quotation marks are removed before execution.
/.../ or \...\	Regular expression quotation characters: Normally used to enclose regular expressions. Take the entire string literally, including the quotation characters—the / or \ characters are *not* removed. Variable substitutions and command substitutions occur. '...', "...", and ∂ have their usual meanings—however, they are not removed.

Single quotation marks, double quotation marks, and ∂ are removed before parameters are passed to programs (unless they are themselves enclosed in quotation marks). For example, here are two ways you might define an AddMenu that compiles a C program in an active window:

Wrong: AddMenu Extras "C Compile" C "{Active}"
Right: AddMenu Extras "C Compile" 'C "{Active}"'

The first example won't work because the {Active} variable will be expanded when the menu is *added* (it should be expanded when the menu item is *executed*). The second example is correct—when the AddMenu command is executed, the single quotation marks defeat variable expansion; they are then stripped off before the item is actually added. The double quotation marks remain in case the pathname of the active window happens to contain any special characters.

◆ *Note:* When quoting spaces (as in filenames), you'll usually use double quotation marks (soft quotes) to permit variable and command substitution.

Slashes (or backslashes) are used to pass regular expressions as parameters to commands, without filename expansion occurring. For example,

```
Search /proc≈/ Sample.p
```

This command searches the file Sample.p for any string beginning with the characters "proc". (See "Pattern Matching" in Chapter 6 and the description of the Search command in Part II.)

■ **Table 5-9**   Special escape conventions

Symbol	Escape convention
∂c	Escape character: Take the single character c literally. The four escape conventions that follow are exceptions to this rule.
∂Return	∂Return is discarded, allowing you to continue a command onto the next line.
∂n	Inserts a return character.
∂t	Inserts a tab character.
∂f	Inserts a form feed character.

How commands are interpreted

When you send text to MPW's command interpreter (by pressing the Enter key or the equivalent), the following sequence of steps is performed:

1. *Alias substitution.*

2. *Evaluation of control constructs.* (This means that control constructs can't be produced by command substitution but can have aliases.)

3. *Variable substitution, command substitution.* All variables (unquoted or quoted with "...", /.../, or \...\ ) are replaced with their value. All commands enclosed in `...` (unquoted or quoted with "...", /.../, or \...\ ) are replaced with their output. If the ellipsis character (Option-semicolon) is found, Commando is executed and the command is replaced by the output of Commando.

4. *Blank interpretation.* After variables and commands have been substituted, the command text is divided into individual words separated by blanks. A blank is an unquoted space or tab.

   ◆ *Note:* The following symbols are normally considered separate words, whether or not they are set off by blanks:

   ; | | | &&   ( )   <   >   >>   ≥   ≥≥   ...

   Within expressions (used with If and Evaluate), all operators are considered separate words, unless they are quoted. See "Structured Commands" in this chapter.

5. *Filename generation.* A word that contains any of these unquoted characters

    ?   ≈   [*   +   «

    after variable substitution is considered a filename pattern. The word is replaced with an alphabetically sorted list of the filenames that match the pattern. (If no filename is found that matches the pattern, an error results.)

6. *Input/output redirection.* Because this step is performed last, variable substitution, command substitution, and filename generation can all be used to form the filenames used in I/O redirection.

7. *Execution.*

You can suppress any part of this process by using quotation symbols as described in the previous section. Remaining single and double quotation marks are removed prior to execution.

◆ What went wrong?

If you ever wonder why a command line doesn't work, refer back to this section
to study the order of command interpretation. You may use the {Echo} variable to
examine how the Shell is interpreting your command. Use the command

```
Set Echo 1
```

With {Echo} defined, the command lines will be echoed to standard output after
they are interpreted by the Shell.
The command Parameters is also useful for finding out which parameters will be
passed to the command. Parameters writes its parameters to standard output.
This command is especially handy when you want to experiment with quoting. For
example, try the following commands:

```
Parameters ≈
        # parameters will be all the files in current directory
Parameters "≈"
        # parameter will be the ≈ character
Parameters "{Commands}"
        # Enclosed in soft quotation marks, the
          variable will be expanded
Parameters '{Commands}'
        # Enclosed in hard quotation marks, the parameter
          will be the string {Commands}
Parameters `date`
        # the output of date will be passed as multiple
          parameters
Parameters "`date`"
        # the output of date will be passed as one parameter ◆
```

Structured commands

Structured commands (listed in Table 5-10) override the normal sequential execution of commands. They can be used interactively and within scripts. They may be nested to any depth, subject to a limitation on stack space. The entire structured command is read before execution begins. All structured commands are built into the MPW Shell.

▲ **Warning**   After the Shell "executes" an opening parenthesis or the opening word of a Begin, If, For, or Loop command, it will not execute any subsequent commands until a matching closing parenthesis or End word is encountered. While it is waiting for the end of the command, the status panel of the Worksheet window will contain the left parenthesis character, (, or the command name. You can abort the entire structured command by typing Command-period. ▲

The status value for a structured command is the status of the last command executed within the structured command (except for the Exit command, which lets you set your own status value).

Expressions (used in If, Break, Continue, and Exit) are defined later in this chapter.

Command	Description
(*command...*)	Parentheses are used to group commands for conditional execution, pipe specifications, and input/output specifications.

Begin...End	`Begin`
	   command...
	`End`
	
	Like parentheses, Begin and End group commands for conditional execution, pipe specifications, and input/output specifications.

If...Else...End	`If` *expression*
	   command...
	`[Else If` *expression*
	   command... `]` ...
	`[Else`
	   command... `]`
	`End`
	
	The command If...Else...End executes the commands following the first *expression* whose value is true (that is, nonzero and non-null). At most one of the lists of commands is executed. If none of the commands is executed, If returns a status value of 0.

For...End	`For` *name* `In` *word...* *command...* `End` The command For...End executes the enclosed commands once for each word from the "In *word...*" list. For each iteration, a variable of the form { *name* } represents the current value from the *word...* list.

Loop...End	`Loop`
	   command...
	`End`
	
	This command repeatedly executes the enclosed commands. The Break command is used to terminate the loop.

Break	`Break [ If` *expression* `]` The command Break terminates execution of the immediately enclosing For or Loop. If the expression is present, the loop is terminated only if the expression evaluates to true (nonzero and non-null).

(Continued)

Command	Description
Continue	`Continue` [`If` *expression*] The command Continue terminates this iteration of the immediately enclosing For or Loop and continues with the next iteration. If the expression is present, the `continue` is executed only if the expression evaluates to true (nonzero and non-null).
Exit	`Exit` [*number*] [`If` *expression*] The command Exit terminates execution of the script in which it appears. If *number* is present, it is returned as the status value of the script; otherwise, the status of the last command executed is returned. If the expression is present, the script is terminated only if the expression evaluates to true (nonzero and non-null). (You can also use Exit interactively to terminate execution of all previously entered commands.)

The return characters in the command definitions above are significant; a return must appear at the end of each line, as shown above, or it must be replaced by a semicolon (;).

The following keywords are recognized when they appear unquoted as the first word of a command:

```
Begin   For   If   Else   Loop   End   Break   Continue   Exit
```

The keyword "In" is recognized when it appears unquoted following For; the keyword "If" is recognized when unquoted following Else, Break, Continue, and Exit. These keywords are not considered special in other contexts and need not be quoted.

◆ *Note:* These keywords cannot be produced as a result of variable substitution or command substitution.

You can apply conditional execution (&& and | |), pipe specifications (|), and input/output specifications (<, >, >>, ≥, ≥≥, Σ, and $\Sigma\Sigma$) to entire structured commands (that is, to Begin...End, If...Else...End, For...End, and Loop...End, and to commands within parentheses).

The operator should appear following the End or closing parenthesis. For example, you can collect the output of a series of commands and redirect it as follows:

```
Begin
   Echo Good day
   Echo Sunshine
End > OutputFile
```

Input/output specifications are discussed later in this chapter. Each of the structured commands is described in detail in Part II.

Control loops

The For and Loop commands are used for looping.

The For...End command executes the enclosed commands once for each word in the "In *word...*" list. The current *word* is assigned to variable *name*, so you can reference the current word by using the Shell variable notation {*name*}. For example,

```
For File In ≈.c
   C "{File}" ; Echo "{File}" compiled.
End
```

The Loop command provides unconditional looping, so you'll need to use the Break or Exit commands to terminate the loop. You can use the Continue command to continue with the next iteration.

For example, the script below runs a command several times, once for each parameter:

```
### Repeat - Repeat a command for several parameters ###
#
#       Repeat command parameter…
#
#   Repeat command once for each parameter in the parameter
#   list. Options can be specified by including them in
#   quotes with the command name.
#
Set cmd "{1}"
Loop
   Shift
   Break If {#} == 0
   {cmd} "{1}"
End
```

In this example, the Shift command (explained in the next section) is used to step through the parameters, and the Break command ends the loop when all the parameters have been used. Using the script Repeat, you could compile several C programs, with progress information, using the command

```
Repeat 'C -p' Sample.c Count.c Memory.c
```

Processing command parameters

In addition to the commands introduced in Table 5-10, there are several other commands that are highly useful in scripts. You can use the following commands to display or modify parameters:

Echo [*parameters...*]

> Writes its parameters, separated by blanks and terminated by a return, to standard output.

Parameters [*parameters...*]

> Writes its parameters, including its name, to standard output. One parameter is written per line, preceded by the parameter number in braces and a space. A return is written following the last parameter.
>
> For example:
>
> ```
> Parameters 1 2 "3a 3b"
> ```
>
> will output
>
> ```
> {0} parameters
> {1} 1
> {2} 2
> {3} 3a 3b
> ```

Shift [*number*]

> Renames the parameters by subtracting *number* from the parameter number; that is, parameters *number* +1, *number* +2, and so on are renamed 1, 2, and so on. If *number* is not specified, the default value is 1. The variables {1}, {2}...{n}, {#}, {Parameters}, and {"Parameters"} are all affected. Shift does not affect parameter {0} (the command name).

See the Hints box "What Went Wrong?" in the previous section, "How Commands Are Interpreted," for some suggestions on using Echo and Parameters to troubleshoot reluctant command lines. For an example of how the various structured commands can work together, see "Sample Scripts" at the end of this chapter.

Expressions

Expressions are used in the If, Break, Continue, and Exit commands. They're also used in the Evaluate command, which returns the result of an expression.

Table 5-11 lists the expression operators in order of decreasing precedence. Some operators have more than one representation; these equivalent symbols are listed on a single line. Groupings indicate operators of the same order of precedence.

■ **Table 5-11**   Expression operators in order of decreasing precedence

Operator		Operation
1. (*expr*)		Expression grouping
2. -		Unary negation
~		Bitwise negation
!	NOT ¬	Logical NOT
3. *		Multiplication
+	DIV	Division
%	MOD	Modulus division
4. +		Addition
-		Subtraction
5. <<		Shift left
>>		Shift right
6. <		Less than
<=	≤	Less than or equal to
>		Greater than
>=	≥	Greater than or equal to
7. ==		Equal
!=	<> ≠	Not equal
=~		Equal pattern (regular expression)
!~		Not equal pattern (regular expression)
8. &		Bitwise AND
9. ^		Bitwise XOR
10. |		Bitwise OR
11. && AND		Logical AND
12. || OR		Logical OR

All operators group from left to right. You can use parentheses to override the operator precedence. Null or missing operands are interpreted as zero. The result of an expression is always a string representing a decimal number. Relational operators return the value 1 when the relation is true and the value 0 when the relation is false.

Logical operators: The logical operators !, NOT, ¬, &&, AND, | |, and OR interpret operands of value 0 or null as false; and they interpret nonzero, non-null operands as true.

Numbers may be

- decimal

- hexadecimal, beginning with either $ or 0x

- octal, beginning with zero

- binary, beginning with 0b

Every expression is computed as a 32-bit signed value. Overflows are ignored.

String operators: The operators ==, !=, =~, and !~ compare their operands as strings. All others operate on numbers.

◆ *Note:* The {CaseSensitive} variable does not apply to the string operators.

Comparing text patterns: The =~ (equal pattern) and !~ (not-equal pattern) operators are like == and != (which compare two strings), except that =~ and !~ are used for comparing a string with a text pattern. The right side is a regular expression against which the left-side operand is matched. For example:

```
If "{1}" !~ /≈.[acp]/
      Echo Filename must end with .a, c, or .p
End
```

◆ *Note:* The regular expression in the above example must be enclosed in the regular expression quotation symbols, /.../. See Chapter 6 for more information about regular expression syntax.

If the regular expression contains the tagging operator ®, then, as a side effect of evaluating the expression, Shell variables of the form {®*n*} containing the matched substrings are created for each tag operator in the expression. (For an example, see the implementation of a wildcard rename command under the description of the Rename command in Part II.)

Use of special characters: Within expressions in the If, Break, Continue, Exit, and Evaluate commands, the following Shell operations are disabled:

- Filename generation

- Conditional execution (I I and &&)

- Pipe specifications (I)

- Input/output specifications (>, >>, ≥, ≥≥, <, Σ, and ΣΣ)

This allows the use of many expression operators that would otherwise have to be quoted. In the case of If commands, the conditional execution or I/O specification should come after the End word. For other commands that contain expressions, you can specify conditional execution or I/O redirection by enclosing the command in parentheses. For example,

```
(Evaluate {1} + {2}) ≥ Errors
```

Redirecting input and output

All built-in commands, scripts, and tools are provided with three open files: standard input, standard output, and diagnostic output (Figure 5-2). By default, standard input comes from the console (the window where the command is executed). Standard output and diagnostics are returned to the console immediately following the command.

■ **Figure 5-2**   Standard input and output
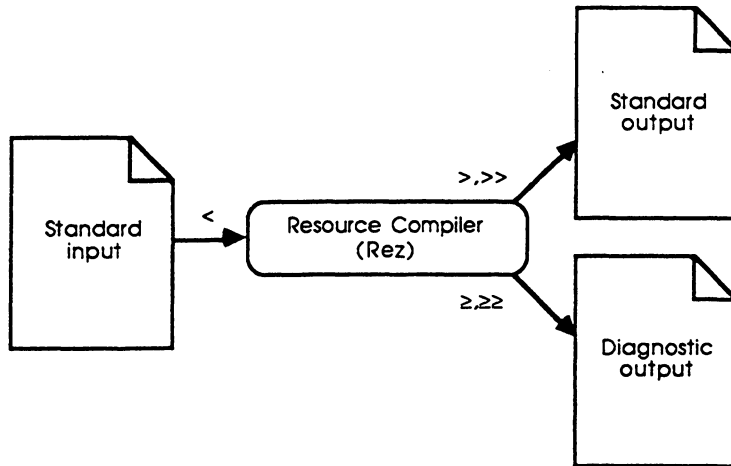
You can override these default assignments with the <, >, >>, ≥, ≥≥, Σ, and ΣΣ symbols described in Table 5-12. Note that input and output specifications are interpreted by the Shell; they are not passed to commands as parameters. You can use parentheses (or the Begin and End commands) to group commands for input/output specifications.

■ **Table 5-12**   I/O redirection

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

Files and windows are treated identically; when given a name, the system looks first for an open window. Input and output can also be applied to selections:

- § indicates the current selection (in the target window).

- *name.§* indicates the current selection in window *name.*

From the point of view of a command running within the Shell environment, input always comes from the standard input file and output goes to the standard output file. The command doesn't need to know whether standard input happens to be text from a file, a window, or a selection, or is typed in from the keyboard. For example, in the statement

```
Program > OutputFile
```

the string "> OutputFile" is interpreted by the Shell and is not passed as a parameter to the command—this process is completely invisible to the command.

I/O specifications also apply to scripts. The standard input, standard output, and diagnostic output files provided to a script become the defaults for commands in the script.

In addition to the sections later in this chapter, you'll find more on input and output in "Standard I/O Channels" in Chapter 12.

Standard input

By default, standard input is supplied by typing text and pressing Enter, or by selecting text that is already on the screen and pressing Enter. You can redirect standard input with the < operator. Note, however, that most commands that read standard input also accept a filename parameter. For example, the following two commands have the same result:

```
Catenate < Sample.c
Catenate Sample.c
```

The Alert command reads from standard input if no message is supplied as a parameter to the command, but Alert doesn't accept filenames as parameters. Thus input redirection is the only way to cause Alert to read input from a file.

```
Alert Errors       # Display Alert box containing the word Errors
Alert < Errors     # Display Alert box containing the contents
                   # of the file Errors.
```

Many commands, including the Assembler and compilers, optionally read standard input to allow input to be read from a pipe (|) or entered interactively, as explained in the next section.

Terminating input with Command-Enter

Many commands read from standard input if no filename is specified. For example, if you execute the command

```
Asm
```

the Assembler will begin reading from standard input—that is, you can enter text to standard input, and the Assembler will process each line as you enter it.

You can repeatedly enter text to a program that reads standard input by typing or selecting text and pressing Enter. You indicate end-of-file by holding down the Command key and pressing Enter (or Command-Shift-Return). For example, after you execute the command

```
Catenate >> {Worksheet}
```

the Catenate command will be running (its name will appear on the status panel at the bottom of the window). You can now enter data from the keyboard or select and enter text from various windows, and all of it will be concatenated to the Worksheet window. Pressing Command-Enter indicates end-of-file and terminates the command.

◆ **Power techniques using standard input**

There are many ways you can save time and effort by running tools from standard input.

For example: Suppose you want to compose a file from parts of other files—and there are ten sections that you want to use in your new file. Normally you'd cut and paste ten times. However, you could also run Catenate from standard input with the command

```
Catenate >> MyComposedFile
```

While Catenate is running, you can open the different source files, select the desired sections, and press Enter. Each time you press Enter, that selection is appended to the file MyComposedFile. When you have finished, press Command-Enter.

Many times it's convenient to quickly type a few lines of code in the Worksheet and then interactively compile (or Rez, using the resource compiler) those lines to test out syntax or examine compiler behavior. You'll find that you can speed up many tasks and increase confidence with quick tests by running tools interactively and using selections as input. ◆

Standard output

By default, **standard output** appears in the window in which the command was executed (that is, the console) immediately following the command. When commands are executed from menus, standard output appears following the selection in the active window. You can redirect standard output with the > and >> operators. For example, the Catenate command

```
Catenate File1 File2 > CombinedFile
```

concatenates File2 to File1—but instead of appearing in the active window, output is sent to the file named CombinedFile. If the window CombinedFile is open on the desktop, its contents are overwritten. Otherwise, the file CombinedFile is replaced (or created if it doesn't exist).

The >> operator appends standard output to the end of a selection, window, or file. If the named file doesn't exist, a new file is created. For example,

```
Catenate § >> AFile
```

appends the contents of the current selection in the target window to AFile. (If the command was entered in the active window, the current selection is the selection in the target window.) You can also specify a selection in a named window:

```
Catenate Sample.c.§ >> AFile
```

Diagnostic output

By default, a command's diagnostic output also appears immediately after the command, interleaved with standard output. The diagnostic output of commands executed from menus appears following the selection in the active window. You can redirect diagnostic output exactly as you redirect standard output, except that you use the operators ≥ (to replace *filename*) and ≥≥ (to append to *filename*) in place of > and >>. You may find it useful to have all error reporting appear in a separate window set aside for that task. For example, in Figure 5-3, the Assembler has been run and error and progress information has been appended to a window called "Errors."

■ **Figure 5-3** Redirecting diagnostic output

```
 File   Edit   Find   Mark   Window   Project   Directory   Build
┌─────────────────────────────────────────────┐
│        HD:MPW:Worksheet                      │
│ ┌───────────────────────────────────┐        │
│ │ Asm -p sample.a >> Errors         │        │
│ └───────────────────────────────────┘        │
│                                              │
│ ┌───────────────────────────────────┐        │
│ └───────────────────────────────────┘        │
├══════════ HD:MPW:Examples:AExamples:Errors ══════════┤
│ ...including HD:MPW:Interfaces:AIncludes:QuickEqu.a    ⇧│
│ ...continuing with sample.a                            │
│ ...including HD:MPW:Interfaces:AIncludes:SysEqu.a       │
│ ...continuing with sample.a                            │
│ QUICKDRAW                                              │
│ GLOBALDATA                                             │
│ SETUPMENUS                                             │
│ SHOWABOUTMEDIALOG                                      │
│ DOCOMMAND                                              │
│ SAMPLE                                                 │
│                                                        │
│ Elapsed time: 17.56 seconds.                           │
│                                                        │
│ Assembly complete - no errors found.  4719 lines.     ⇩│
│ MPW Shell  ◁□                                       ▷▣ │
└─────────────────────────────────────────────┘
```

Often it is useful to redirect both standard output and diagnostic output to the same file, using the summation operators Σ (to replace *filename*) or ΣΣ (to append to *filename*). The example used in Figure 5-3 might then be written in the Worksheet like this:

```
Asm -a Sample.a ΣΣ SampleTest
```

Then both the output of Sample.a and its diagnostics, including any errors, would be appended to a file named HD:MPW:AExamples:SampleTest.

Pseudo-filenames

Pseudo-filenames are a set of device names that you can use in place of filenames; however, they have no disk files associated with them. Any command can open a pseudo-filename as a file. These device names are most commonly used for I/O redirection.

Table 5-13 shows the available pseudo-filenames.

■ **Table 5-13** Pseudo-filenames

Pseudo-filename	Description
Dev:Console	Always refers to the current console device. The console is the default source of input and the default destination of output—that is, the active window where a command is entered and its output displayed.
Dev:Null	Null device. If you read from Dev:Null, it immediately returns end-of-file. If you write to Dev:Null, output is thrown away.
Dev:StdIn	Standard input.
Dev:StdOut	Standard output.
Dev:StdErr	Diagnostic output.

Pseudo-filenames are especially useful inside a script if you want to do something like sending standard output to the diagnostic output of the script. Here are some examples:

```
Echo "An error message." >> Dev:StdErr
Echo "HELP !" >> Dev:Console
```

Dev:Null is useful in scripts when you want to throw away diagnostic output. For example:

```
Eject 1 ≥ Dev:Null
```

This command ejects the disk in drive 1; if no disk is in drive 1, the script continues to run silently. (Note that you would also need to set {Exit} to 0—see "Variables" earlier in this chapter.)

Editing with the command language

Almost all menu commands have equivalents in the command language. In most respects, there is no difference between the menu commands and their command-language equivalents. The primary difference is that with the command language, you enter commands in the active (frontmost) window, while an editing command acts on a selection in another window. You can explicitly name a window as a parameter to the command. If you don't specify a window, the command acts on the target window.

For example, to use command-language techniques to edit the file Sample.a, you must first open that file, and then click on another window, such as the Worksheet window, to make it the active window. You enter your commands in the active window, as shown in Figure 5-4. When you select text in the active window, it's highlighted in the normal Macintosh fashion. In other windows, selected text is indicated by dim highlighting (outlining), as shown in the target window in Figure 5-4.

■ **Figure 5-4**   Text highlighted in the active and target windows
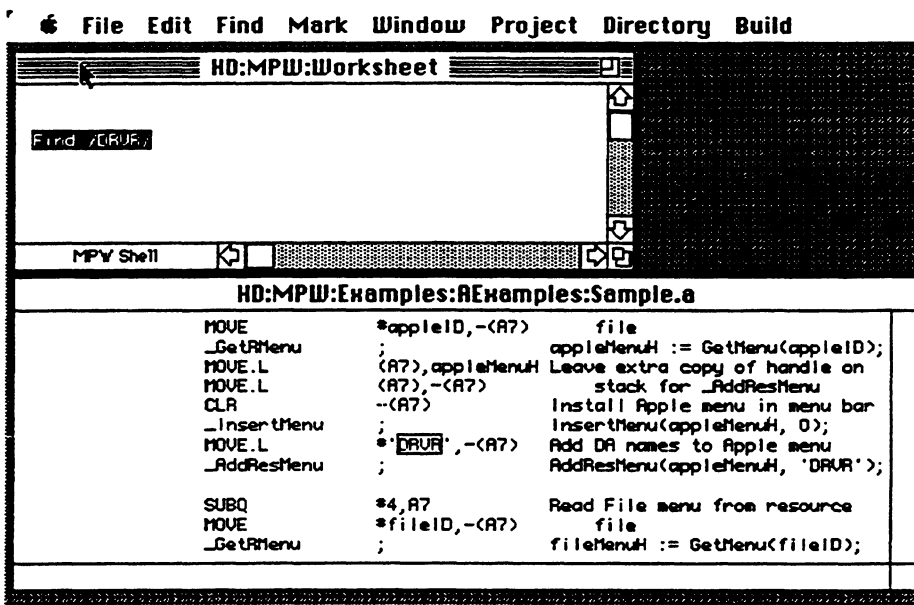
Editing commands generally act on a selection. (The Find command simply creates a selection—"DRVR" in this example.)

The § metacharacter (Option-6) is the current selection character. It signifies the current selection in a window. For example, the following command erases from the current selection or insertion point in the target window to the end of the window:

```
Clear §:∞
```

The infinity character, ∞ (Option-5), is a selection operator that indicates the end of a window, as described in Chapter 6. For interactive editing, press Command-Delete to clear to the end of a file.

Defining your own menu commands

The AddMenu and DeleteMenu commands are for adding and deleting menu items. The AddMenu command takes three parameters: the menu name, the item name, and the command text. For example,

```
AddMenu Find 'Top of Window/U' 'Find • "{Active}"'
```

This command adds a "Top of Window" item to the Find menu, using the keyboard equivalent Command-U. When you select the menu command, the corresponding commands are executed. (The Top of Window item moves the insertion point to the top of the active window.)

Invoking a user-defined menu Command is the same as entering the command text from a window—variable substitution and command substitution are performed normally. Note, however, that the text of the menu command is processed twice—once when the AddMenu command itself is executed, and again whenever the menu item is executed. This means that you have to be especially careful in your use of quotation symbols. The mysteries of quoting are explained earlier in this chapter in "Quoting Special Characters," together with further AddMenu examples. You should also pay particular attention to the section "How Commands Are Interpreted." For further information, and more examples, see the AddMenu command in Part II.

Sample scripts

The following examples use most of the Shell's features to illustrate how you can extend the MPW Shell with your own commands.

"AddMenuAsGroup"

The following script adds an extra feature to the AddMenu command:

```
#       AddMenuAsGroup - AddMenu, grouping user defined menu items:
#
#       AddMenuAsGroup [ menuName [ itemName [ command ]]]
#
#       AddMenuAsGroup duplicates the functionality of the AddMenu
#       command, adding a disabled divider before the first user-
#       defined menu items in the File, Edit, and Find menus.
#
Unalias
Set Exit 0
Set CaseSensitive 0
If ({#} == 3) AND ("{1}" =~ /File/ OR "{1}" =~ /Edit/ ∂
 OR "{1}" =~ /Find/)
        If `AddMenu "{1}"` == ""        # If this is the first addition
                                        # in {1},
            AddMenu "{1}" "(-" ""       # add the group divider
        End
End
AddMenu {"Parameters"}
```

When adding menu items to the predefined menus, it's useful to add a disabled dotted line item to separate the new menu items from the original ones. The script above automatically adds the separator before the first new item in the File, Edit, and Find menus, the only predefined menus that can be modified by using AddMenu. If you put this script in a file named AddMenuAsGroup, the following alias will override the built-in AddMenu command:

```
Alias AddMenu AddMenuAsGroup
```

"CC"

The following script extends the C command by making it possible to compile a number of specified files:

```
#       CC - Compile a list of files with the C compiler
#
#       CC [options…] [file…]
#
#       Note that the options and the files may be intermixed, and
#       that all options apply to all the files. The individual C
#       commands are echoed to diagnostic output as they are executed.
#
Unalias
Set Exit 0
Set CaseSensitive 0
Set options ""
Set files ""
Set exitStatus 0
Loop
        Break If {#} == 0
        If "{1}" =~ /-[diosu]/          # options with a parameter
                Set options "{options} '{1}' '{2}'"
                Shift 2
        Else If "{1}" =~ /-≈/           # other options
        Set options "{options} '{1}'"
        Shift 1
        Else
        Set files "{files} '{1}'"
        Shift 1
        End
End
For i in {files}
        C {options} "{i}" || Set exitStatus 1
End
Exit {exitStatus}
```

Chapter 6 Advanced Editing

MPW'S EDITING OPERATIONS ARE AVAILABLE AS BUILT-IN COMMANDS, including scriptable selections and the use of regular expressions. These commands enable powerful find-and-replace functions and make it possible to automate editing operations by using scripts.

Menu commands for editing are described in Chapter 3. The basics of routine interactive editing are described in Chapter 4. For a full description of the use of the command language, see Chapter 5. Appendix B contains a summary of selections and regular expressions. ■

Contents

Editing commands

The command language contains editing commands that duplicate the functions of many of the menu commands and provide additional capabilities. The editing commands are listed in Table 6-1. (They're explained in detail in Part II.)

■ **Table 6-1**   Built-in editing commands

Command	Description
Adjust [-c *count*] [-l *spaces*] *selection* [*window*]	Adjust lines in a selection.
Align [-c *count*] *selection* [*window*]	Align text with first line of selection.
Clear [-c *count*] *selection* [*window*]	Delete selected text.
Copy [-c *count*] *selection* [*window*]	Copy selected text to the Clipboard.
Cut [-c *count*] *selection* [*window*]	Copy selected text to the Clipboard and then delete the selection.
Find [-c *count*] *selection* [*window*]	Find and select text.
Format [*option...*] [*window ...*]	Set or view font name, font size, tabs, and indents on specified windows.
Mark [-y l -n] *selection name* [*window*]	Assign the marker *name* to the range of text *selection* selected in *window*.
Markers [-q] [*window*]	Print list of all markers associated with *window*.
Paste [-c *count*] *selection* [*window*]	Replace *selection* with the contents of the Clipboard.
Position [-c l -l] [*window ...*]	Display the position of the selection in each specified window.
Replace [-c *count*] *selection* *replacement* [*window*]	Replace *selection* with *replacement*.
Revert [-y] [*window...*]	Revert window to last saved state.
Target *name*	Make a window the target window.
Undo [*window*]	Undo last command.
Unmark *name...* *window*	Remove the marker(s) *name...* from the list of markers available for *window*.

If no *window* parameter is specified, editing commands act on the **target window**, which is the second window from the front. Therefore, to edit the active window, you'll need to switch to another window for entering your commands, or else specify the name of the active window in the command line. (The Target command makes a window the target window; the Shell variables {Active} and {Target} always contain the full pathnames of the current active and target windows.)

Most editing commands take the following parameters:

-c *count*   You can specify a repeat count with the -c option; *count* is the number of times the command should be executed. *Count* may also be the infinity character, ∞ (Option-5), which specifies that the operation should be repeated as many times as possible.

selection   Most editing commands act on a selection, either the current selection in the target window or another selection that you specify. First, an implicit Find is done to select the specified text. Then the text is modified. The selection syntax is defined in the next section.

window   The optional *window* parameter lets you specify the name of the window to be affected by a command without changing the position of the affected window.

A command modifies the selection only if there were no syntactic errors in the selection and if all regular expressions were matched. Commands run silently unless an error occurs.

In addition to the routine editing commands incorporated in the command language, MPW includes a number of tools and scripts that are useful for many specialized editing tasks. Some of these are listed in Table 6-2. See Part II for detailed information.

■ **Table 6-2**   MPW tools useful for editing

Editing tools	Description
Canon [*option...*] *dictionaryFile* [*inputFile...*]	Replace a file's identifiers with canonical spellings given in *dictionaryFile*.
Compare	Compare text files.
Entab [*option...*] [*file...*]	Convert runs of spaces to tabs.
FileDiv [*option...*] *file*	Divide a file into several smaller files.
Line [*number*]	Find specified line number in a file.
MatchIt [*option...*] [*window*]	Match currently selected left language delimiter with its mate in *window*.
RezDet [*option...*] *resourcefile*	Detect inconsistencies in resources.
Translate [*option...*] *source* [*destination*]	Convert selected characters.

Selections

A **selection** is a parameter to editing commands; it tells the command what text to select. A selection may be any of the following:

■ A line in a file (selected by line number)

■ A position in a file

■ A marker

■ A specific character pattern

■ A selection that begins and ends with any of the above

As an example of the selection syntax, consider the definition of the Find command:

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

Find takes a selection as an argument and selects the argument text (or sets the insertion point). An actual command might take the form

Find /shazam/

This command finds and selects the first instance of the string "shazam" that appears after the current selection. (The slashes are used to enclose a *pattern*, a special case of a selection, as explained below.) No count is specified, so the command is executed once. No window name is specified, so the command operates on the target window.

Table 6-3 shows all of the selection operators. These are more fully explained in the sections following the table.

■ **Table 6-3**   Selection operators

Operator	Type of selection
Current selection	
§	Current selection in the target window (§ is Option-6 on the keyboard)
Line numbered selections	
n	Line number *n*
!*n*	Line number *n* lines after the end of the current selection
¡*n*	Line number *n* lines before the start of the current selection (¡ is Option-1)
Position (insertion point)	
•	Position before the first character of the file (• is Option-8)
∞	Position after the last character of the file (∞ is Option-5)
Δ*selection*	Position before the first character of *selection* (Δ is Option-J)
*selection*Δ	Position after the last character of *selection*
selection!*n*	Position *n* characters after the end of *selection*
selection¡*n*	Position *n* characters before the beginning of *selection*
Pattern (characters to be matched)	
/*pattern*/	Pattern (regular expression)—search forward (see "Pattern Matching," below)
pattern\\	Pattern—search backward
Extended selection	
selection1:*selection2*	Both selections and everything in between
marked selection *name*	The *name* of a marked selection may contain any characters except
	§ ! ¡ (: • ∞ Δ / \\
Grouping	
(*selection*)	Controls order of evaluation

A formal definition of selections can be found in Appendix B.

All of the operators group from left to right, and evaluation proceeds from left to right. The selection operators are listed below in order of precedence:

/ and \	Everything within slashes is taken as a regular expression and is evaluated as explained below under "Pattern Matching."
( )	Controls the order of evaluation.
Δ	Indicates position.
! and ¡	Indicates position (! = after; ¡ = before).
:	Joins two selections.

Some examples will illustrate why it's important to pay attention to the precedence of these operators:

```
Δ/begin/!1          means              (Δ/begin/)!1
                    rather than        Δ(/begin/!1)
```

That is, the insertion point is located after the "b" of "begin" rather than after the "n."

```
/begin/:/end/!1     means the selection    /begin/:(/end/!1)
                    rather than the position  (/begin/:/end/)!1
```

That is, the character after "end" is included in the selection, as shown in Figure 6-1.

■ **Figure 6-1**   A selection specification
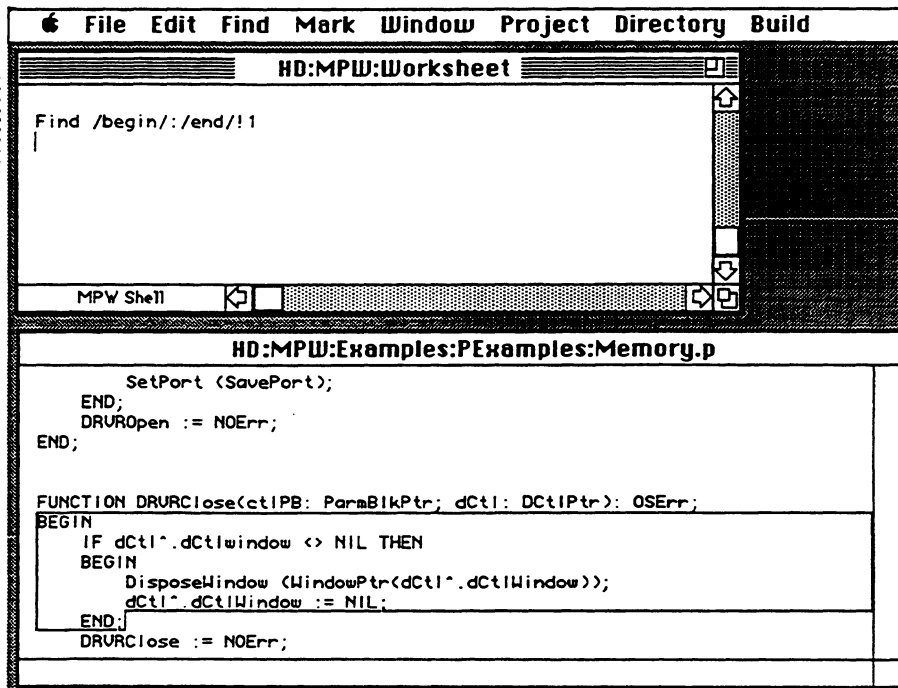
Current selection (§)

The current selection character, § (Option-6), always indicates the current selection in a window. If no window is specified, § indicates the current selection in the target window. For example, consider the windows shown in Figure 6-2.

■ **Figure 6-2**   Selections in two windows

```
 ┌──────────────────────────────────────────────────────────────┐
 │  🍎  File  Edit  Find  Mark  Window  Project  Directory  Build │
 │ ┌────────────────────────────────────────────────────┐        │
 │ ══════════════════ HD:MPW:Worksheet ═══════════════ ▣ │        │
 │ │                                                   ⇧ │        │
 │ │ Replace § ∂n│                                       │        │
 │ │                                                     │        │
 │ │                                                     │        │
 │ │                                                   □ │        │
 │ │                                                   ⇩ │        │
 │  MPV Shell    ◁□▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▷⊡ │        │
 │ ┌────────────────────────────────────────────────────────────┐│
 │ │            HD:MPW:Examples:PExamples:Memory.p              ││
 │ │   vRect:      Rect;                                         ││
 │ │   myWindow:   WindowPtr;                                    ││
 │ │ BEGIN                                                      ││
 │ │     IF dCtl^.dCtlWindow = NIL THEN                         ││
 │ │     BEGIN                                                  ││
 │ │         GetPort (SavePort);                                ││
 │ │             myWindow := GetNewWindow(RsrcID(dCtl),nil,POINTER(-1));││
 │ │             windowpeek(myWindow)^.WindowKind := dCtl^.dCtlRefNum;  { show a []││
 │ │         dCtl^.dCtlWindow := myWindow;     { let the desk manager know too }││
 │ │         heapGrow := MaxMem (heapGrow);│                    ││
 │ │         SetPort (SavePort);                                ││
 │ │     END;                                                   ││
 │ │     DRVROpen := NOErr;                                     ││
 │ │ END;                                                       ││
 │ └────────────────────────────────────────────────────────────┘│
 └──────────────────────────────────────────────────────────────┘
```

The command

```
Replace § ∂n
```

would replace the current selection in the target window with a single return (newline) character. ("∂n" is a special code for inserting a return—see "Inserting Invisible Characters" later in this chapter.)

Note that the current selection is a dynamic quantity—it's determined by the last *subexpression* evaluated and thus represents the current state of a selection as it's being calculated. For example, consider the command

```
Find /if/:§!1:§!1
```

At various points in the evaluation of the search string "/if/:§!1:§!1", the current selection (§) has the following different values:

Before calculation	The pre-existing selection in the target window
After "/if/"	"if"
After "/if/:§!1"	All characters from "if" to (and including) the first character after the "if"
After "/if/:§!1:§!1"	All characters from "if" to (and including) the first two characters after the "if"

Selection by line number

If you give a number unquoted by slashes as a selection, it is taken to be a line number. This may be an absolute line number or a number of lines relative to the current selection. For example, to select line 3 of a file, you'd use the command

```
Find 3
```

This expression is equivalent to

```
Find '3'
```

but

```
Find 3   or   Find '3'
```

is not equivalent to

```
Find /3/   or   Find \3\
```

The exclamation mark and inverted exclamation mark (! and ¡) specify the number of lines after or before the current selection. Thus, the command

```
Find !3
```

selects a line that is 3 lines beyond the current selection. Note that the !n notation specifies a line relative to the *end* of the current selection (that is, n lines past the line containing §∆); ¡n specifies a line relative to the *start* of the current selection (n lines before the line containing ∆§).

Position

A **position** is a special case of selection. Position means the location of the insertion point only. The Δ character (Option-J) is used to convey position relative to a selection. For example, consider the commands

```
Find 3
Find Δ3
Find 3Δ
```

The first Find command *selects* the entire third line in the target file. The Find Δ3 and Find 3Δ commands *place the insertion point* at the beginning and at the end of the third line, respectively.

You can also use the ! and ¡ operators to specify a position that's a given number of characters from a selection: *selection!n* specifies a position *n* characters after *selection,* and *selection¡n* specifies a position *n* characters before *selection.*

Note that this leads to two different uses of the ! and ¡ operators, as in the following example:

```
Find !4!4
```

The first "!4" indicates a *selection* that's 4 *lines* beyond the current selection; the second "!4" indicates the *position* that's 4 *characters* beyond the end of that selection.

You can specify other positions in a file with the following special notation:

- • (Option-8)   Position preceding the first character in a file
- ∞ (Option-5)   Position following the last character in a file

Markers

A marker is a selection that has been given a name. A marker may be used as a selection variable. You can mark as many selections and insertion points as you wish. You can create markers directly by selecting text in a window and then clicking the Mark command in the Mark menu. See "Mark Menu" in Chapter 3 for more information on the interactive use of markers. This section describes the general behavior and programmatic use of markers.

Behavior of markers

Markers may be as simple as a position in a window, but more often a marker names a range of positions. Markers have the special attribute of being able to remember their assigned position(s) even when you're making editing changes all around them. For example, typing before marked text has the effect of moving both the text and its associated marker toward the end of the window. Editing "inside" the range of a marker will either increase or decrease the range of the marker, depending on whether the editing was an insertion or deletion, respectively.

Markers are "sticky." For example, if an insertion point is marked and you enter text at that point, everything you type will be added to that marker.

If you delete the text encompassing a marker the marker will also be deleted. For example, if the string "xyz" is deleted and the character "y" is marked, the "y" marker will be deleted. However, if the string "xyz" itself is marked as "y," deleting the string "xyz" will result in marker "y" being reduced to an insertion point.

Markers are associated with individual windows. When you switch between windows, the Mark menu is updated to reflect the markers of the new active window.

Markers are persistent. They are saved in the resource fork of the file you are editing, just like font, tab, and other information about the window. However, markers are not saved to the Clipboard. Thus, if you cut a marked region and paste it somewhere else, the marker will be lost.

Markers are case sensitive. A marker named "Y" is different than a marker named "y."

Programmatic use of markers

You can create or delete Markers programmatically by using the following three Shell commands:

Mark [-y | -n] *selection name* [*window*] Assign the marker *name* to range of text
                                            selection selected in *window*.

Markers [-q] [*window*]   Print a list of all markers associated
                                              with *window*.

Unmark *name*...window·   Remove the marker(s) *name*... from the
                                               list of markers available for *window*.

For example, to mark the currently selected text in the target window with the name "Function B" and to replace any previous marker of that name, you would type

```
Mark -y  §  'Function B'
```

The new marker name will appear in the Mark menu. You might remove the marker later by using the Unmark command:

```
Unmark  'Function B'  "{Target}"
```

This command would remove that marker from the target window.

To use markers as selections, just type the marker name. For example,

```
Find  george
```

For further details on the Shell commands for markers, see Part II.

◆ Automatic Selection

You'll find many ways to use markers for automatic selections. For example, to automatically select the output of a script for a user, you could use a script similar to this:

```
Mark  §∆ X    #Mark the start of output
Make          #Run your Make command
Find  X       #Select the output of Make ◆
```

Pattern

A **pattern** may be either a literal text pattern or a regular expression (defined in the next section). You specify a pattern between the /.../ and \...\ delimiters. Forward slashes indicate a search forward, and backslashes indicate a search backward. A forward search begins at the end of the current selection and continues to the end of the file. A backward search begins at the start of the current selection and continues to the beginning of the file. For example, the command

```
Find /myString/
```

searches forward for the literal expression "mystring." (Recall that to specify case-sensitive pattern matching, you need to set the Shell variable {CaseSensitive}, or select the "Case Sensitive" menu item.)

Extending a selection

A colon is used to join two selections. For example,

```
Find /begin/:/end/
```

This command selects "begin," "end," and everything in between. (See Figure 6-1.) Compare this command with

```
Find /begin≈end/
```

which looks for a begin-end pair on a single line.

Pattern matching (using regular expressions)

Regular expressions are a shorthand language for specifying text patterns. Regular expressions are used in editing commands, in the Search command (which searches one or more files for occurrences of a pattern), and in If and Evaluate expressions following the =~ and !~ operators. Most of the regular expression operators may also be used in filename generation.

Regular expressions are always used within the pattern delimiters /.../ or \...\.

A special set of metacharacters, called regular expression operators, is used in regular expressions (and in filename generation). The regular expression operators are listed in Table 6-4.

■ **Table 6-4**   Regular expression operators

Operator	Meaning
c	Any character matches itself (unless it's one of the special characters listed below)
∂ *c*	Defeat the special meaning of the following character (*c* is taken literally) except ∂n = return ∂t = tab ∂f = form feed
'...'	Literalize enclosed characters
"..."	Literalize enclosed characters, except ∂, {, and `
?	Any single character (other than a Return)
≈	Any string of 0 or more characters that does not contain a Return
[*character*...]	Any character in the list
[¬*character*...]	Any character not in the list (¬ is Option-L on the keyboard)
*regularExpr**	Regular expression 0 or more times
regularExpr+	Regular expression 1 or more times
regularExpr«*n*»	Regular expression *n* times (« is Option-\; » is Option-Shift-\)
regularExpr«*n*,»	Regular expression *n* or more times
regularExpr«*n*₁,*n*₂»	Regular expression n_1 to n_2 times
(*regularExpr*)	Grouping
(*regularExpr*)®*n*	Tagged regular expression (where $0 \leq n \leq 9$)
*regularExpr*₁*regularExpr*₂	*regularExpr*₁ followed by *regularExpr*₂
•*regularExpr*	Regular expression at the beginning of a line
regularExpr∞	Regular expression at the end of a line

These characters are considered special in the following circumstances:

∂	Special everywhere except within single quotation marks ('...')
? ≈ * + [« ( )	Special anywhere except within [...], '...', and "..."
®	Special only after a right parenthesis,)
•	Special as the first character of an entire regular expression
∞	Special as the last character of an entire regular expression
/ \	Special if used to delimit a regular expression
¬	Special only after a left bracket, [
-	Special in brackets, except immediately following a left bracket, [

Their precedence (from highest to lowest) is as follows:

1. ( )

2. ? ≈ * + [ ] « ®

3. concatenation

4. • ∞

A formal definition of regular expressions can be found in Appendix B. The rest of this section describes the use of regular expressions for describing selections.

Character expressions

In the simplest case, regular expressions consist of literal characters enclosed in slashes. For example,

```
/what the ?/
```

Notice one complication, however: if the literal character happens to be one of the regular expression operators (such as "?"), it will be specially interpreted rather than taken as a literal character. If you want to specify a literal character that happens to have a special meaning within the context of regular expressions, you'll have to precede it with the escape character, ∂, or enclose it in quotation marks. The character ∂ has the effect of "literalizing" the character that follows it. For example, to find the literal expression given above, you would use one of the following commands:

```
Find /what the ∂?/
Find /what the '?'/
Find /'what the ?'/
```

You could also use double quotation marks, that is "...".

Wildcard operators

In addition to literal characters, regular expressions can include the operators ?, ≈ (Option-X), and [ ], which are used as follows:

?	Any character other than a Return
≈	Any string not containing a Return, including the null string (this is the same as ?*)
[*characterList*]	Any character in the character list (as defined below)
[¬ *characterList*]	Any character *not* in the list

These operators are also used as wildcards in filename generation. (You can also use the *, +, ?, ≈, [...], and «...» operators in filename generation—see "Filename Generation" in Chapter 5.)

A character list is an expression consisting of one or more characters enclosed in brackets ([...]). It matches any character found in the list. The case sensitivity of characters in the list is governed by the {CaseSensitive} variable. A list may consist of individual characters or a range of characters, specified with the minus sign (–). For instance, the following two commands are equivalent:

```
Find /[ABCDEF]/
Find /[A-F]/
```

You can also mix the two notations:

```
Find /[0-9A-F$]/
```

◆ *Note:* This command specifies any of the characters 0 through 9, A through F, and $. To specify the] or – character, place it at the beginning of the list or literalize it with the escape character, ∂.

The negation symbol, ¬ (Option-L), lets you specify any character *not* in the list. For example,

```
Find /[¬A-Z]/
```

This example specifies all characters *except* the letters A through Z. (To specify the ¬ character itself, place it anywhere in the list other than the beginning, or literalize it by preceding it with the escape character, ∂.)

Repeated instances of regular expressions

The asterisk character (*) matches zero or more occurrences of the immediately preceding regular expression. The plus sign (+) matches one or more occurrences of an expression. For example, the command

```
Find /[0-9]+/
```

will find any string of one or more digits.

You can also specify an expression that occurs an explicit number of times by using the «n» notation:

regularExpr«n» Regular expression n times
regularExpr«n,» Regular expression at least n times
regularExpr«n_1, n_2» Regular expression at least n_1 times and at most n_2 times

For example,

```
Replace -c ∞ /' '«4,»/ ∂t
```

This command finds any string of four or more spaces and replaces it with a tab.

(The -c ∞ option specifies a repeat count of "infinity"; that is, it replaces all occurrences of of the selection to the end of the document.)

Tagging regular expressions with the ® operator

The ® (Option-R) operator tags a regular expression between parentheses. This operator is useful with the Replace command, for example, in reformatting tables of data. Consider a table with two columns of numbers separated by spaces or tabs:

```
123        456
123        456
123        456
123        456
```
...and so on

The following Replace command switches the order of the two columns, which are separated by one tab:

```
Replace  -c  ∞   /([0-9]+)®1[ ∂t]+([0-9]+)®2/   '®2 ®1'
```

Translated into English, this expression means

`[0-9]+`	Match one or more characters in the set "0" to "9".
`([0-9]+)®1`	Remember that selection (the expression enclosed in parentheses) as ®1.
`[ ∂t]+`	Next, match at least one space or tab.
`([0-9]+)®2`	Then match one or more characters in the set "0" to "9" and remember it as ®2.
`'®2 ®1'`	Finally, replace the whole matched string with what was remembered as ®2, a space, and what was remembered as ®1.

◆ *Note:* The quotation symbols are stripped off, as explained under "Quoting Special Characters" in Chapter 5.

After this sequence is executed, the table will look like this:

```
456        123
456        123
456        123
456        123
```
...and so on

Matching a pattern at the beginning or end of a line

In the context of regular expressions, the • metacharacter (Option-8) means that the subsequent expression must be matched at the beginning of a line. For example, the regular expression

```
/•main/
```

will match a line that begins with "main" but not a line that begins with "*space* main". The beginning of a line is either the first character after a return or the first character of the file.

Likewise, the ∞ metacharacter (Option-5) means that the previous expression must be matched at the end of a line. The regular expression

```
/main∞/
```

will match a line that ends with "main" but not a line that ends with "main *space* ". The end of a line is either the last character of a line prior to the return, or the end of the file.

Notice that • and ∞ have another meaning within selections. Within a pattern, they indicate the beginning and end of a *line*. Within a selection, they indicate the beginning and end of the *file*.

Inserting invisible characters

You can use the Shell escape character, ∂, to insert the following special characters in text:

∂n   return
∂t   tab
∂f   form feed

For more information on the escape character, see "Quoting Special Characters" in Chapter 5.

Note on forward and backward searches

Forward and backward searches aren't always completely symmetrical. For example, consider the command

```
Find /?*/
```

This command finds zero or more occurrences of any character other than a return. The first time you execute this command, some range of characters will be selected if the current selection is not at the end of a line. However, in subsequent invocations, the selection will stick at the end of the line and only an insertion point will be left at the end of the line. This is because the * metacharacter matches zero occurrences and the search starts with the character following the current selection—in this case, the insertion point preceding a return. A backward search of the form

```
Find \?*\
```

will never stick at the beginning of a line. This is because a backward search begins with the first character to the left of the current selection and so has the effect of jumping over a return after encountering it.

◆ **Solving selection difficulties**

What If a selection expression doesn't select what you Intended? Ask yourself questions like these:

■ Am I quoting special characters?
For example, the
`∙(∙`
character is special. If you are searching for this character, then you must use
`∙∂(∙`

■ Do I remember the definitions of special characters?
Review the special character definitions in Appendix B.

■ Are my precedence and usage correct?
Consider the slightly different syntax of these two Find commands:

`Find ∙:/main/`

This tells MPW to select everything from the beginning of the file until the first occurrence of the word ˋmainˊ

`Find /∙main/`

This tells MPW to select the next occurrence of the word ˋmainˊ at the beginning of a line.

■ Do the individual pieces select what I intended?
Break the difficult expression down into small parts. Try each part separately to make sure that it does what you want. Then add each new, tested part to create more complicated expressions. ◆

Some useful examples

This section shows some examples of the complex use of regular expressions.

Transforming DumpObj output

The DumpObj command, described in Part II, formats the contents of an object file. This example shows how to transform a DumpObj listing, such as the following, back into valid assembly code.

```
000000:   4EBA 06F8        'N...'   JSR       *+$06FA           ; 6004282A
000004:   4EBA 04EA        'N...'   JSR       *+$04EC           ; 60042620
000008:   3B7C 0014 FCC4 ';|....'   MOVE.W    #$0014,$FCC4(A5)
00000E:   266D 0010        '&m..'   MOVEA.L   $0010(A5),A3
000012:   2653             '&S'     MOVEA.L   (A3),A3
000014:   0C5B 0000        '.[..'   CMPI.W    #$0000,(A3)+
000018:   6600 0008        'f...'   BNE       *+$000A           ; 60042152
00001C:   3A1B             ':.'     MOVE.W    (A3)+,D5
00001E:   6600 0010        'f...'   BNE       *+$0012           ; 60042160
...and so on
```

You could position the insertion point at the beginning of the code and use the following Replace command:

```
Replace -c ∞ /?«41»/ "∂t∂t"     # replace everything up to the
                                # instruction with 2 tabs
```

However, the previous command works only because DumpObj happens to place the instruction at column 42. The following example, by defining some Shell variables, works regardless of the exact column layout:

```
Set hex   "[0-9A-F]«4,6»"        # 4 to 6 characters in the set 0-9 and A-F
Set space "[ ∂t]+"               # 1 or more spaces or tabs
Set chars "∂∂'?+∂∂'"             # 1 or more of any character between  ∂
                                 # single quotes
Replace -c ∞ /{hex}:({space}{hex})«1,3»{space}{chars}{space}/ "∂t∂t"
```

Finding a whole word

The following example illustrates how you could find an exact match for a C identifier that you had previously defined in the variable {ident}:

```
Set tokensep "[¬a-zA-Z_0-9]"   # a token separator is any character
                               # not in the set a-z, A-Z, _, or 0-9
Set CaseSensitive 1            # set to "true"—the case of each
                               # character must match
```

The following Find command is not quite right, because it selects not only the matched identifier but also the token separator on each side of the identifier:

```
Find /{tokensep}{ident}{tokensep}/
```

The following Find command selects only the matched identifier. It accomplishes this by adding 1 to the starting position of the selection (Δselection!1), and uses that as the starting point for a new selection that extends to the beginning of the next token separator:

```
Find Δ/{tokensep}{ident}{tokensep}/!1:Δ/{tokensep}/
```

◆ Bulldozer

If you are making a very large number of changes (such as scripted global replacements in a large file), that file's memory may become fragmented. If this happens, the rarely seen bulldozer icon may replace your cursor. The bulldozer tells you that MPW is trying to clear more memory space for your file.

If you regain the regular cursor, you can close (save) the window and reopen it, thus completely reinitializing its memory area. If the bulldozer lingers it may mean that your computer will be busy with this one script over the weekend. To avoid this problem, it is better to reboot and modify your script to proceed in stages so that you don't run out of file memory.

For example, let's suppose that you have a 10,000-line file that you wish to edit with this script:

```
clear -c ∞ /•   /
replace -c ∞ /"("/  "["
```

These two formatting commands operate on each of the 10,000 lines in the file, a total of 20,000 operations (assuming that each line was changed). Unless you have a gigantic amount of RAM, this is probably more work than your computer can comfortably handle. (Of course, you might also try to free more memory by turning off MultiFinder or increasing MPW's application area, but doing so would help only a little in this case.)

If your Macintosh has 5 MB of RAM, then you might be able to perform the first 10,000 operations of the first command without ever glimpsing the dread bulldozer. In this case, modify the script so that after the first command the script closes the window (thus automatically saving the file) and reopens it to continue with the next large operation:

```
clear -c ∞ /•   /
close -y {MyEditWindow}
open  -t {MyEditWindow}
replace -c ∞ /"("/  "["
```

Alas, what if the bulldozer appears during the execution of the first command? First deduce at what point the bulldozer appeared (let's say somewhere well after the first 4000 lines were changed) and then modify your script to stop processing at regular intervals. In the example that follows, the editing operation stops after 4000 operations, closes the window (thus saving it automatically), and reopens it. Then the program returns to the top of the file and resumes editing for another 4000 operations.

```
find•
  loop
    clear -c 4000 /•   /
    break if {status} ≠ 0
    close -y {MyEditWindow}
    open  -t {MyEditWindow}
  end                                                        ◆
```

Chapter 7   Projector: Project Management

PROJECTOR IS A BUILT-IN MPW FACILITY FOR MANAGING PROGRAMMING PROJECTS of any size. Projector makes it easy to keep track of the revision history of the files comprising your programs: who changed what, when, why, and other information. You can use Projector to create experimental branches of a project and later remerge the successful efforts.

The syntax of all Projector commands is summarized at the end of this chapter. You can find detailed information and examples for each of these commands in Part II. There are a number of Projector-specific terms defined throughout this chapter; these terms can also be found in the glossary. ∎

Contents

About Projector

Projector is a collection of built-in MPW commands and windows that help programmers (both individuals and teams) control and account for changes to *all* the files (documentation, source, applications, and so on) associated with a software development project. Use Projector to coordinate changes among a team of programmers and to maintain a history of project revisions. When you begin work on a project, you select the appropriate project and check out the files needed just as books are checked out from the public library—although in this case, Projector distributes both read-only and modifiable copies of its "books."

Projector requires the presence of MPW 3.0 and does not run outside MPW as either a Macintosh application or desk accessory. The terms and concepts introduced in this section are discussed in greater detail the the section "Components of a Project," near the end of this chapter.

Overview

During the evolution of a software development project, each team member invariably makes numerous changes to the source and documentation files. Sometimes the changed source files are alternative versions or experimental efforts; later you want to discard the failed efforts and merge the best versions together. Projector is designed to substantially ease this task by providing an easy-to-use yet powerful facility for file management that is valuable to both the individual programmer working on a small project and to a team of programmers working on a complex set of programming projects. Use Projector to organize your files into projects that can be stored locally on a hard disk, a 3.5-inch disk, or remotely anywhere on the AppleTalk network.

A **project** is a conceptual entity for organizing files, analogous to an HFS directory. Once within Projector, each file becomes a **revision tree**. Each revision tree comprises the entire historical sequence of revisions and branches of a particular file. Any of these revisions may be opened for reading only or checked out exclusively by one user for modification. Figure 7-1 shows how three files might appear as three revision trees in a hypothetical project. The sequentially numbered circles represent revisions. Those circles with letter suffixes are branches, which may in turn sprout their own branches and subsequent revisions. The numbering scheme for revisions, branches, and revisions of branches is explained in the section "Revisions" later in this chapter.

■ **Figure 7-1**   A project structure
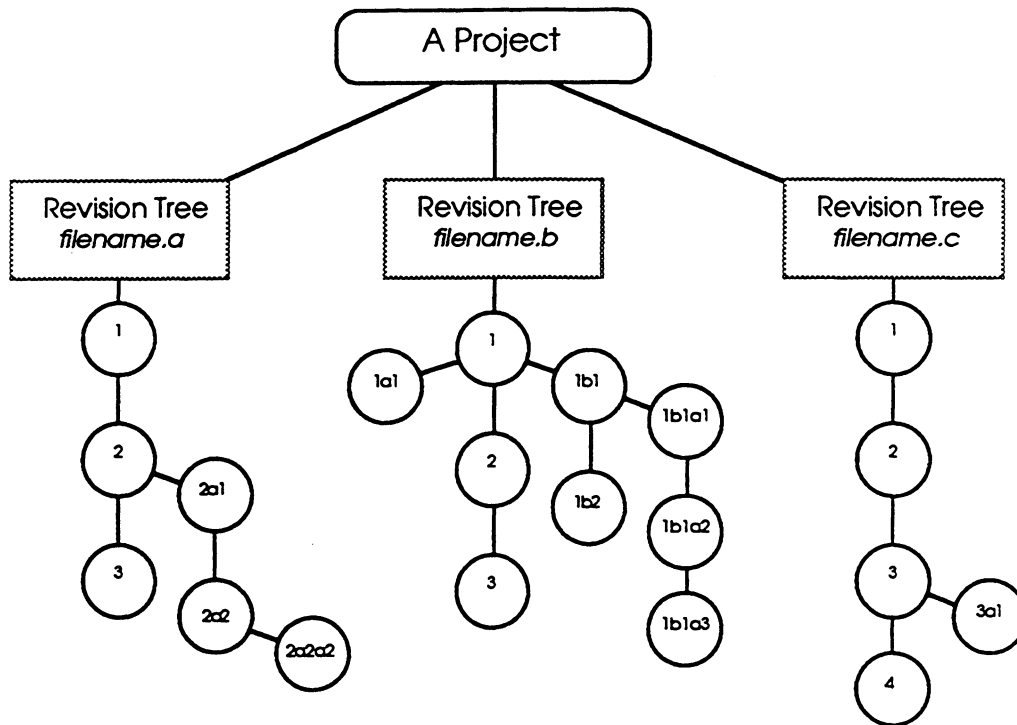
When you check out a "file" for modification you are actually checking out a copy of a revision—usually the latest revision—from the file's revision tree. The revision you have checked out appears in your HFS directory as an ordinary file named after its associated Projector revision tree. When you check it back in, the "file" becomes the next revision in its Projector revision tree.

When checking out a file for modification you can write a comment describing the changes you're about to make (so other project users can see why you have checked out the revision). Projector remembers that the revision is checked out and denies access to anyone else attempting to modify checked out revisions. (Of course, you can always create a new branch off a checked-out revision.)

You can check the revision back into Projector at any time, although you would normally check in revisions as soon as your modifications are complete and tested. Once your revision is checked back in, the next sequential number is, by default, appended to its name to identify its place in the revision tree. This revision is now available to anyone on the team.

Besides supporting a single sequence of revisions to each file, Projector also allows alternative revisions to be created. This feature is called revision **branching**. Branching makes possible

- the modification of old revisions

- work on the same revision of a file by several programmers simultaneously

- parallel, experimental lines of development

See the section "Creating Branches" later in this chapter.

Whenever you go through the simple check-in process, you are encouraged to document all the changes you have made and the reasons for these changes. This allows the project's current status and history to be easily retrieved by all team members. It's also extremely handy when you have to go back through old revisions to find a problem or retrieve something of value.

Projects may contain other projects, called **subprojects**. This last fact is of key importance, because it lets you break down large projects into subunits that can still be accessed as a whole by those outside the immediate programming team. See "Nested Projects" later in this chapter.

Features

Some of Projector's key capabilities are listed here:

- Projects and subprojects can be organized into a hierarchy.

- All revisions to a file are saved in the revision tree. Each revision is uniquely identified by its filename and revision number.

- Nontext files as well as text files may be stored in the project.

△ **Important**   Be careful of programs that may inadvertantly delete Projector's `'ckid'` (that is, check ID) resources from files. When a program such as Microsoft Word™ saves a file, it deletes the file's `'ckid'` resource. These resources contain the identification Projector uses to track files. △

- Revisions made to text files are stored in a compact format.

- Access by multiple users is supported. Requests to modify the Project database are controlled by user name on a per-project basis. AppleShare can be used to assign privileges.

- A flexible naming facility allows revisions to be identified by symbolic name as well as by filename and revision number. (See "Symbolic Names" later in this chapter.)

- The entire history and status of all revision trees in the project can be displayed conveniently and accurately. A data field for comments is saved with revisions, revision trees, and projects. Projector also associates another data field, called Task, with every revision of a file.

- Projector supports a command line interface so that you can embed Projector commands in MPW Shell scripts.

- A window-based interface is provided for convenient and easy browsing and access to projects. Any or all Projector windows can be opened from the Project menu. (However, not all command-line functions are supported in the window interface.)

- Scripts that compare and merge revisions are supplied in `"{MPW}Scripts:"`

Limitations

Keep these rules in mind when using Projector:

- All files (revision trees) in a project must have unique names.

- There is no easy, integrated way to change a filename.

- Revisions cannot be arbitrarily deleted out of sequence. (See "Deleting Revisions" later in this chapter.)

- Revisions to nontext files are not compressed.

- Commas are not allowed in filenames.

- Symbolic names must not be hyphenated.

See "Project Administration" later in this chapter for information on ways to get around some of these limitations.

Using Projector: A walk-through

This section takes you through each step of the principle operations of Projector, demonstrating the functions of the principal windows. The concepts and organization will become evident as you go through this hands-on tutorial.

First, you'll create a new project. Next, you'll check a new file into the project you just created. Finally, you'll mount the project Sample and check out revisions from the project Utilities.

The concepts used here are defined in detail in the section "Components of a Project" and the summary sections following this walk-through. You can also find Projector terms in the glossary.

Creating a new project

The simplest way to create a project is to use the New Project window. Follow these steps to create a project called Test:

1. Double-click the MPW Shell icon to launch MPW and open the Worksheet.

   ◆ *Note:* Make sure that the {User} variable is set to your name.

2. Type
   ```
   Set User 'username'
   ```
   Note that the Chooser user name will be used if it is available.

3. Set your directory to the Projector example folder by typing the following command in the worksheet:
   ```
   Directory "{MPW}Examples:Projector Examples:"
   ```

4. Select New Project from the Project menu. The New Project window appears. You can display all of the other Projector windows in a similar fashion (see Figure 7-2).

■ **Figure 7-2**   New Project window

```
┌─────────────────────────────────────────────────────────┐
│ □▨▨▨▨▨▨▨▨▨▨▨▨▨▨ New Project ▨▨▨▨▨▨▨▨▨▨▨▨ │
├──────────────────────┬────────────────────────────────────┤
│      ⊂⊃ Maui         │ Project Name: ┌──────────────────┐  │
│ ┌──────────────────┐ │               │                  │  │
│ │⊖ Projector Examples│ │ User: Jeff Parrish              │  │
│ ├──────────────────┤⟨⟩│ New Project comment:             │  │
│ │□ sample          │▲│ ┌───────────────────────────┐⟨⟩  │  │
│ │                  │ │ │                          │▲│  │  │
│ │                  │ │ │                          │ │  │  │
│ │                  │ │ │                          │ │  │  │
│ │                  │ │ │                          │ │  │  │
│ │                  │▽│ │                          │▽│  │  │
│ └──────────────────┘ │ └───────────────────────────┘    │  │
│      ( Open )        │                                    │  │
│ ┌────────┐ ┌───────┐ │                  ╔═══════════════╗│  │
│ │ Drive  │ │ Eject │ │                  ║  New Project  ║│  │
│ └────────┘ └───────┘ │                  ╚═══════════════╝│  │
└──────────────────────┴────────────────────────────────────┘
```
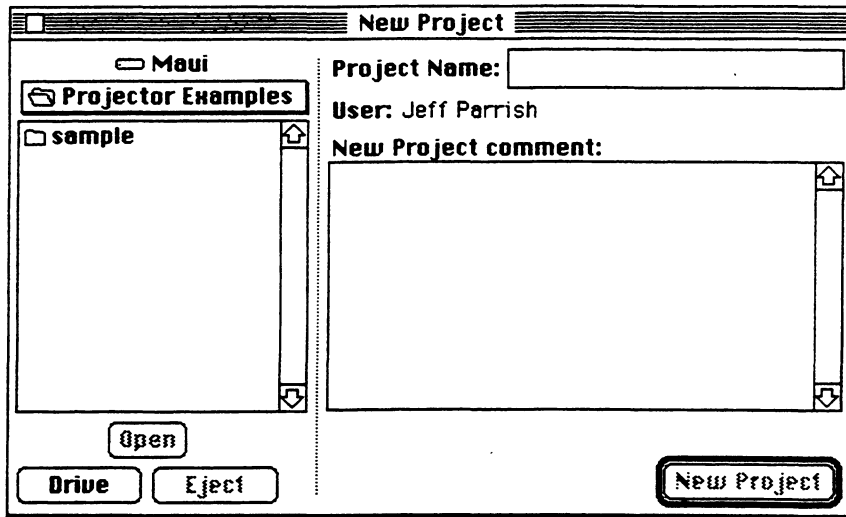
5. Type the new project's name, Test, into the Project Name field of the New Project Window.

6. You may add some descriptive information about the new project in the Comment field.

7. Finally, click the New Project button at the lower right corner of the window. (If the user name is not specified, the New Project button remains disabled.)

The New Project window should now look like the example shown in Figure 7-3, except that your name will appear in the User field.

■ **Figure 7-3**   New Project window after creating a project
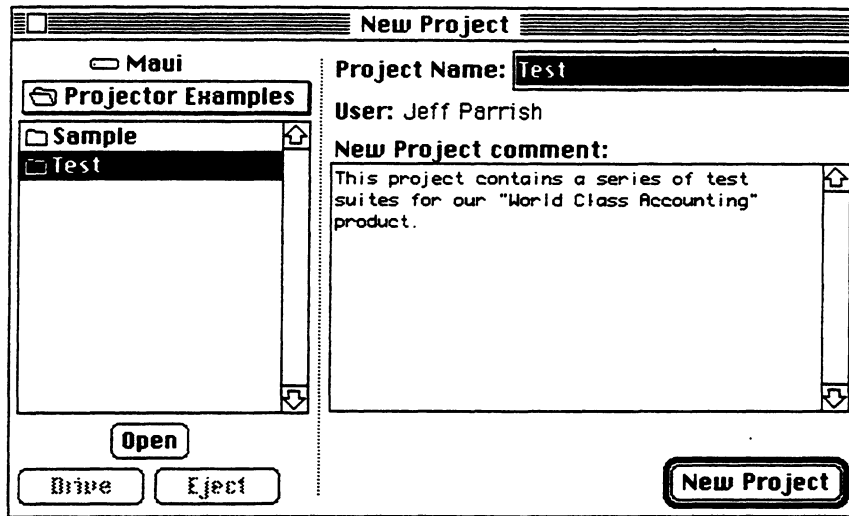
The left side of the New Project window is a list similar to that in the Standard File dialog; it displays the Macintosh's HFS file structure. This lets you create a project anywhere in the file system, either under an existing project or in some other directory.

You could also create the new project Test by using the command line

```
NewProject Test
```

This creates a project named Test whose project directory, created by Projector, is :Test:. Projector maintains all information regarding this project in the file ProjectorDB within this directory. Nested projects will appear as folders within this directory. The checkout directory is set to the current directory at the time of the check out. (See the description of checkout directories later in this chapter.)

Test is automatically mounted for you and also becomes the current project. Test can actually be an HFS pathname or, if a project is currently mounted, a Project pathname. In either case, the name of the new project is the leaf of the path. If an HFS path is given, that directory becomes the project directory for the new project. If a project pathname is given, the new project becomes a subproject of its parent, as shown here:

```
NewProject Sample⌿Fortran
```

This creates a new project, Fortran, which is a subproject of the Sample project. If the project directory of the Sample project were

```
"{MPW}Examples:Projector Examples":Sample:
```

then the project directory of the Fortran project would be

```
"{MPW}Examples:Projector Examples":Sample:Fortran
```

This command is equivalent to the previous example:

```
NewProject "{MPW}Examples:Projector Examples":Sample:Fortran
```

Since the project directory of the Sample project is "{MPW}Examples:Projector Examples:Sample", the Fortran project automatically becomes a subproject of the Sample project.

When you mount a project, all of its subprojects are mounted at the same time. See the section "Components of a Project" later in this chapter for more information on project directories and details of naming conventions.

Checking in a revision

When you have finished modifying an existing revision of a file, you should check it back into the project. When you want to add a new file to a project, follow the same procedure.

You can add comments to indicate the changes you've made. Projector will record the date and time, as well as other information about the revision. The changes you have made become part of a new revision in the file's revision tree. (Of course, read-only revisions cannot be checked in because they do not contain changes and therefore cannot create new revisions.)

When using the Check In window keep in mind that the HFS directory (displayed in the list on the left side of the window) is similar to a regular Macintosh Standard File dialog that displays the files and folders. The Check In window's list does not show files that are not a part of a project unless you click the Show All Files box.

To check a new file into a project, follow these steps:

1. Execute the following command to create a simple new file that can be used in this example:

   ```
   Echo "a new file" > newfile
   ```

2. From the Project menu, select Check In. The Check In window appears (Figure 7-4).

3. You must check the Show All Files check box at the bottom left of the window to select a new file because the list at the left side of the window, by default, lists only revisions that belong to the current project.

4. Select the file "newfile" in the list at the left of the Check In window.

◆ *Note:* You can select several files at once for check-in. Use shift-click to select contiguous filenames by dragging. Use command-clicking to select discontiguous filenames.

5. Type a comment into the comment field of the Check In window. Whenever you check a revision into a project, it is a good idea to add a comment describing the changes you've made (or in the case of a new file, its purpose).

6. To check "newfile" in with an initial revision of "1.0" (the default would be "1"), click the Revision button. When the Revision Number dialog box appears, type "1.0" into the Revision field, and then click the OK button. The Check In window should now look like the example shown in Figure 7-4.

■ **Figure 7-4**   Check In window
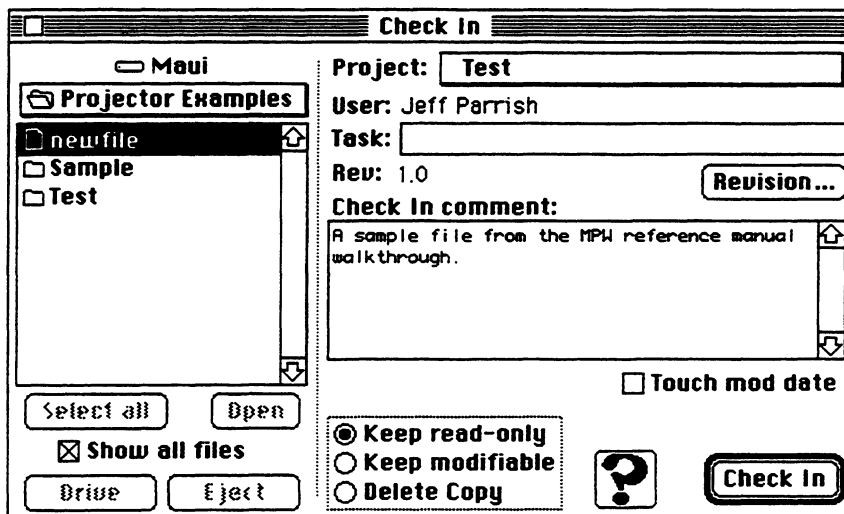
7. Click the Check In button.

Later, when you open the Check Out window, you will see that "newfile" has been entered into the project Test.

◆ Reading icons in the Check In window

This is a list of small icons that may appear in the Check In window's list.

Read-only. The file is a read-only file belonging to the current project.

Modified read-only. The file is a modified read-only file (explained later in this chapter) belonging to the current project.

The **regular document** icon represents a file that does not belong to any project. It is visible only when Show All Files is checked.

The **pencil** icon means that the HFS file is checked out from the current project for modification by the current user.

The **lock** icon means that the HFS file is checked out from the current project for modification by another user.

File belongs to a project other than current project. Appears only in the Check In window when Show All Files is checked.

Modifiable file belonging to another project (denoted by the tiny plus sign in the lower-right corner). Appears only in the Check In window when Show All Files is checked.

Corrupt `ckid` **resource** (explained later in this chapter). Appears only in the Check In window when Show All Files is checked. ◆

Each of the features of the Check In window is discussed below:

Project pop-up

Click on the "Project:" field at the top center of the Check In window to select the current project. This is the project into which you will be checking files. The HFS list (shown on the left side of the Check In window) will be updated to list the selected project's associated checkout directory (if there is one).

User field

The value of the {User} variable is displayed here. The {User} variable must be set in order to check in revisions.

Info (question mark) button

When you click the Info (question mark) button in the Check In window, the right side of the window displays Projector's current information on the selected file (that is, the contents of the file's `ckid` resource). The Check Out window also has an Info View. Figure 7-7 shows the Info View of the Check Out window. If you are checking in a new file, the Info View is blank because Projector has not yet created a `ckid` resource for it. If you are checking in a revision that was checked out for modification, you can modify the Comment or Task fields in the Info View. These changes are saved in the revision's `ckid` resource.

Keep Read-Only, Keep Modifiable, and Delete Copy buttons

The default action after checking in a file is to leave you with a read-only copy. The three radio buttons in Figure 7-4 can alter this default. The top button corresponds to the default; it leaves you with the read-only copy. Use the Keep Modifiable button to check in the file and still retain a modifiable copy. The Delete Copy button deletes your copy of the file once it is successfully checked in. These last two radio buttons correspond to the -m and -delete options of the CheckIn command.

Adding new files to a project

You add new files to projects by checking the Show All Files check box in the Check In window (or by using the -new option of the CheckIn command). When that check box is checked, all files in the current directory are shown. Any files not belonging to a project may then be selected and checked in. These files will be added to the current project.

▲ **Warning**   If, for any reason, the 'ckid' resource of the revision is corrupted or removed, then Projector will not be able to identify the revision, which becomes an orphan file, no longer belonging to any project. If you still need to check the file in, move or rename your copy, cancel the check-out of the revision that is damaged (see "Checking Out a Revision" later in this chapter), check out the revision again, and use the TransferCkid command to move the Projector information from the checked-out revision to your orphaned file. ▲

In the Check In window, you can select a file only if it is currently checked out. In other words, only the enabled filenames can be selected. This restriction means that only files that have been checked out for modification can be checked in. In the Info View, all files are selectable so that you can also get information on your read-only files.

Touch Mod Date check box

This check box appears on both the Check In and the Check Out windows. In both cases it lets you change the convention for time and date stamping. In the Check In window, Projector's default is to leave the date and time of check-in untouched. Check the Touch Mod Date box to stamp the revision of the file that you are checking in with the current date and time, that is, the check-in time.

Changing a revision's revision number

Use the Revision button in the middle right corner of the Check In window to open a dialog box that allows you to specify the revision number for the revision you are about to check in. Besides enabling you to specify the revision number, the Revision dialog box also lets you create a branch. This is useful when you want to save your changes but not along the main trunk of the revision tree.

It is not possible to specify the name of the branch that will be created.

Locating a project

The set of mounted projects defines a set of **project trees**. This list tells Projector the names of the mounted projects and where their project directories are located. If a project is not in one of those trees, the project cannot be accessed. If a root project is moved or renamed (by changing its project directory), users must change their MountProject commands in order to reconnect to the project.

Use the MountProject command to see a list of currently mounted projects.

Checking out a revision

The simplest way to check out a revision is to open the Check Out window (shown in Figure 7-5) by selecting Check Out from the Project menu. You can also use the CheckOut command, as explained later in this section.

Normally, when you begin work in MPW you first check out a revision for modification from a Projector revision tree. The checked-out revision then appears in your directory as a regular HFS file bearing the filename of its Projector revision tree. When you check this file back into its project, it will be saved as the next sequential revision in its revision tree.

◆ *Note:* The Check Out facility in Projector does not copy the actual data if you already have a copy of the revision that you are checking out. For example, if you already have a read-only copy of revision 5 of file.c in hd:work: and you check out a modifiable copy of revision 5 of file.c into that same directory, Projector does not recopy the data of that revision. The 'ckid' resource is updated to reflect the new check-out.

Keep in mind that, unlike the Check In window, the list at the left side of the Check Out window lists the project heirarchy, not HFS.

A checked-out revision matches its corresponding checked-in revision in all ways except for the 'ckid' resource and optionally the modification date. By default, when checking out a revision, the modification date is set to the current time in order to trigger any makefile dependencies. This setting is needed to automatically trigger rebuilds when old revisions of source files are checked out. You can override this default behavior by clicking the Touch Mod Date check box in the Check Out window or by using the -noTouch option to the CheckOut command.

As part of the check-out process, you can leave a comment describing the changes you plan to make. Other members of your group can then see what is being done to the checked-out revisions. This does not prevent anyone else from reading the revisions in any revision tree of a project, but it does prevent anyone else from modifying the same revisions at the same time. After completing your work, check your revision back in with Projector, and add a note describing any changes you have made (if you had not done so when you checked out the revision or if you want to modify your initial comment).

Follow these steps to mount the Sample Project:

1. Click on the worksheet to make it the active window.

2. Mount the sample project by typing this command:
   ```
   MountProject "{MPW}Examples:Projector Examples:Sample"
   ```

3. Set the checkout directories by executing these commands:
   ```
   CheckOutDir -project Sample∫Commands "{MPW}Scripts:"
   CheckOutDir -project Sample∫Utilities "{MPW}Scripts:"
   ```

   This directs Projector to place files from these projects into the Scripts folder.

4. From the Project menu, select Check Out.

5. Find the Utilities project in the Project list at the left side of the window, select it, and click the Open button. The Check Out window appears as shown in Figure 7-5.

   ◆ *Note:* The control at the left of the Check Out window is the Project list and works somewhat like a Standard File dialog, except that it displays only mounted projects and the revision trees and revisions within those projects. It does not show HFS directories.

■ **Figure 7-5**   Check Out window

```
▤□▦▦▦▦▦▦▦▦▦▦▦▦ Check Out ▦▦▦▦▦▦▦▦▦▦▦▦▦
  ┌─────────────────────┬───────────────────────────────┐
  │   Current Project   │ Checkout to: │ Maui:MPW:Scripts: │
  │   ▣ Utilities       │ User: Jeff Parrish              │
  │ ┌─────────────────┐ │ Task: ┌──────────────────────┐ │
  │ │⎕ CheckInActive ⬆││ Check Out comment:              │
  │ │⎕ CheckOutActive ││ ┌────────────────────────┐⬆    │
  │ │                 ││ │                        │     │
  │ │                 ││ │                        │     │
  │ │                 ││ │                        │⬇    │
  │ │              ⬇ ││ Select Files in Name: ☒ Touch mod date │
  │ └─────────────────┘ │ None                            │
  │ ┌────────┐ ┌──────┐ │              Cancel Checkout    │
  │ │Select all││ Open │ │ ◉ Read-only  ┌─────────────┐  │
  │ └────────┘ └──────┘ │ ○ Modifiable  ?  │ Check Out │  │
  │   Select newer      │   □ Branch      └─────────────┘  │
  └─────────────────────┴───────────────────────────────┘
```
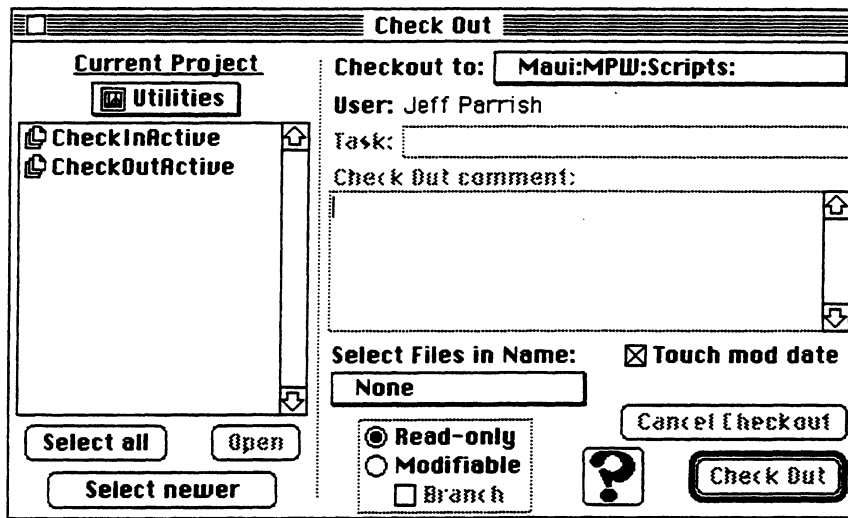
6. Choose the Modifiable radio button.

7. Click the Select All button in the lower-left corner. This selects the latest trunk revisions of all revision trees that are not checked out for modification in the Test project (you have the option of selecting only individual revisions from the project).

8. Click the Check Out button in the lower-right corner.

◆ *Note:* You can automatically open a TEXT revision as you check it out by holding down the Option key while you click the Check Out button.

Now you are ready to open and modify the revisions you have just checked out from the Utilities project. No one else can modify the specific revisions you have checked out until you check them back in by using the Check In window (or CheckIn command).

The procedure for checking out nontext revisions is the same as the procedure for checking out text revisions.

These visual cues may appear in the Check Out window.

A project. A similar, but larger icon is used in the Finder to represent the ProjectorDB file.

A Projector revision tree. Appears only in the Check Out window.

The regular document icon represents an individual revision currently available. It is visible when an individual revision tree is displayed.

When a project is displayed (so that all its revision trees are listed), the pencil icon means that the latest revision of the main trunk is checked out for modification by the current user. When an individual revision within a revision tree is displayed (a list of revisions), the pencil icon means that the particular revision is checked out for modification by the current user.

When a project is displayed (so that all its revision trees are listed), the lock icon means that the latest revision of the main trunk is checked out for modification by another user. When an individual revision within a revision tree is displayed (a list of revisions), the lock icon means that the particular revision is checked out for revision by another user. ◆

Here is an explanation of each part of the Check Out window:

Checkout directory

The "Check out to" field in the window's upper-right corner shows the directory where checked-out revisions will be placed. Clicking the field displays a pop-up menu that gives you three choices:

■ The checkout directory for the current project

■ The current directory

■ Access to a Standard File dialog where you can choose any directory

The directory that is displayed by default in the field is the check out directory for the current project (see CheckOutDir command).

User field

The value of the {User} variable is displayed here. The {User} variable must be set in order to check out revisions.

Task and Comment fields

The Task and Comment fields are optional but it's recommended that you get in the habit of always stating your purpose. Use the Task field to let others know why you have checked out the revision. This information can help later if you have to review a long revision history to find something.

The Comment field is intended to document the specific changes to a revision while the Task field could be used to relate different revisions, perhaps across several files (that is, Projector revision trees). You can change the purpose stated in the Task and Comment fields when you check the revision back in.

For example, implementing a certain feature might require several changes to each of three files. Each revision might have a different comment, but the tasks for all the revisions might say "Enhancement X." The Task field makes it easier to look at the history of a project and determine what changes were made to accomplish various tasks.

To save a Task field across several revisions, select the revisions in the list at the left of the window and type directly into the Task field of the Check In window proper. Then click the Check In button to check in the selected files. To save a unique Comment field with a particular revision, first select that revision and then click the Info button. Type into the Info view's Comment field, click Save, and then click Done.

Select Newer button

Use the Select Newer button to select all the revisions for which the latest revision on the main trunk is not in the "Check out to" directory. Projector looks in the checkout directory to determine which revisions to compare against the project. If you have a modifiable file (or a file that is on a branch) in the checkout directory, then the corresponding revision trees are not selected. In the case of modifiable files, this is done to prevent checking out a file by overwriting a modifiable file. In the case of any files that you may have on a branch, Projector assumes that you want to leave that file alone.

◆ *Note:* If you do have read-only copies of branches, then use the Select All button to ensure that you get the latest revision on the main trunk of every revision tree.

If you hold down the Option key as you use the Select Newer button, then revisions that are new to the project (that is, you don't already have a copy of them) will not be selected.

The Select Newer button cannot be used when checking out revisions for modification.

Select All button

If you are checking revisions out for modification, the Select All button will select all revision trees whose most recent revisions are not already checked out for modification. When checking out read-only copies, the Select All button will select all revisions in the project.

◆ *Note:* The two buttons Select All and Select Newer do not actually check out revisions; they simply make a selection in the Project list (the leftmost frame of the Check Out window). Only the Check Out button (the button in the lower-right corner of Figure 7-5) actually checks out the selected revisions.

Read-Only/Modifiable buttons

The two radio buttons at the bottom of the window specify read-only or write-modify types of revision check out. The default is to check out read-only copies. Everyone with access to the project can check out revisions for reading-only at any time. But if the revision is checked out for modification, no one else can check that revision out for modification.

◆ Modifiable read-only file

On rare occasions you may want to modify a read-only file. For example, suppose you have taken a number of modifiable files home. You may have also brought along certain read-only copies of files that you did not expect to modify. However, once you get into your work at home you discover that you do, after all, need to make changes in these files. In such an exceptional case, you can use Projector's ModifyReadOnly command. In the Worksheet type:

`ModifyReadOnly` *filename*

You can now make changes to this read-only file exactly as if you had checked it out as a modifiable file, with two exceptions:

■ Once it is saved, the special modifiable read-only icon appears next to the filename in the Project list.

■ When you check a modified read-only file back into its project you will have no problem unless someone else has modified the same revision. In this case you must manually merge the two versions. See "Merging Branches" later in this chapter for step-by-step instructions.

Obviously, the ModifyReadOnly command is intended only as an emergency convenience; you should not routinely rely on it. ◆

Branch check box

Use this check box to create a branch. This control is enabled only when doing modifiable check-outs. If the file that you have selected is locked, then this check box will be selected automatically. See the "Creating Branches" section that follows this walk-through for instructions on creating and merging branches. See the "Components of a Project" section for detailed explanations of branching in Projector.

◆ *Note:* You can select dimmed files by holding down the Option key while you click the names.

Touch Mod Date check box

When the Touch Mod Date check box is checked, the revisions checked out (into the check out to directory) are touched, that is, the modification date is set to the date and time of check-out. If the box isn't checked, the checked-out revisions have the same modification date as when the corresponding revision was checked in.

Checking out a particular revision

To check out a particular revision of a revision tree, first display the revision tree by selecting its filename and clicking the Open button (or you can simply double-click the filename). You can then select the revision you want. You can select several noncontiguous filenames by command-clicking.

Info (question mark) button

If you have selected an individual revision in the Project list, clicking the Info button overlays the right side of the Check Out window with the Info View, which is a display of the information pertinent to the selected revision. At this point you can edit the Comment and Task fields. You can also get information about revisions that are checked out for modification or information about any other revision in the project.

Select Files in Name

Click on the Select Files in Name field to display a pop-up menu showing both private and public symbolic names, the private names appearing at the top, the public at the bottom. When you select a symbolic name, the project list at the left side of the window highlights all revision trees that correspond to the selected symbolic name. Click the Check Out button to check out all of them. This makes it easy to select at any time just those revisions that comprise, for example, an alpha release.

◆ *Note:* If you manually change the selection after selecting a symbolic name, you will void the selection of the name.

Discarding changes

To throw away any changes, use the Cancel Checkout button located in the lower-right corner of the window just above the Check Out button. For example, if half the revision trees were accidentally checked out for modification, you could undo the mistake by simply canceling their check outs. This button is also handy if you want to experiment or if you checked out a modifiable copy when you intended to make it read-only.

Using the CheckOut command

Revisions can also be checked out using the CheckOut command:

```
CheckOut file.c -m
```

This will place a modifiable copy of file.c in the checkout directory of the current project. You can change the checkout directory by using the CheckOutDir command. If no project has been mounted, or if file.c does not exist in the current project, Projector reports an error. You can get a modifiable copy by using the -m option; the default behavior of CheckOut is to distribute read-only copies.

There are several different ways to specify where checked-out revisions should be placed. The rules for determining the directory are as follows, from highest to lowest precedence:

1. The directory indicated if a nonleaf name is specified.
2. The directory specified with the -d option.
3. The checkout directory for the project (see the CheckOutDir command).

For example:

```
CheckOut -d hd:MPW: file.c hd:work:defines.h
CheckOut hd:MPW:main.c library.h
```

This first CheckOut will place a copy of file.c in hd:MPW:file.c and a copy of defines.h in hd:work:defines.h. In this case, the checkout directory was not used. The second CheckOut will place a copy of main.c in hd:MPW:main.c and a copy of library.h in the checkout directory for the project.

◆ **A quick switch from Read-Only to Modifiable**

You can quickly check out the active (frontmost) window and change its status from read-only to modifiable by using the CheckOutActive script found in the project

```
{MPW}Examples:Projector Examples:Sample:Utilities:
```

Likewise, you can quickly check in modifiable files by using the CheckInActive script. ◆

Creating branches

In addition to supporting a sequence of revisions to a file in a project, Projector also lets you create branches. **Branches** are alternative sequences of revisions that are parallel to the main revision sequence. Branches may be used for

- the modification of old revisions
- work on the same revision of a file by several programmers simultaneously
- parallel, experimental lines of development

You can create a branch off a revision during the check-out or check-in process by clicking the Branch check box in the Check In or Check Out windows.

Checking out a modifiable copy of an old revision creates a new branch (as shown in Figure 7-6). When file.c is checked back in, it will automatically become Revision 2a1.

- **Figure 7-6**   A changing revision tree
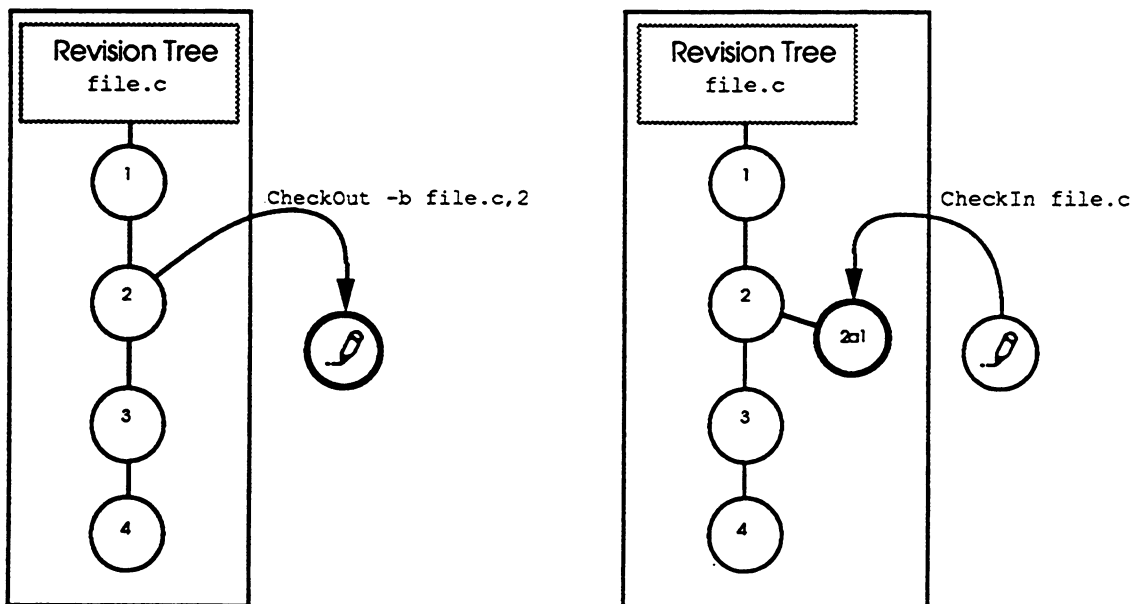
The following command will create a branch when checking in a revision:

```
CheckIn file.c -b
```

In this example, the user did not need to specify a revision number to create a branch. The branch is automatically created off the revision that was checked out. This is possible because Projector remembers which revision was checked out. When a revision (obtained from revision *x*) is checked back in, it can create a revision in one of two places:

- The next revision after *x*, continuing on the same line
- On a branch off revision *x*

Referring to Figure 7-6, you'll see that the user could not check in file.c as Revision 3a1 or Revision 5.

The following command will create a branch and number the first revision 2:

```
CheckIn file.c,2 -b
```

If Revision 4 of file.c was initially checked out, the preceding CheckIn command would create Revision 4a2 (or 4b2 if Revision 4 already had one branch, and so on).

Merging branches

You can merge a branch revision with the trunk by using MergeBranch. You'll find details on this script in Part II.

1. Make sure you have checked out the branch revision you want to merge.
2. Execute the MergeBranch script on the file you want to merge.
   ```
   MergeBranch file.c
   ```

Retrieving information

Information retrieval is one of the most important aspects of any source file control system. Once again, there are two different ways to get information out of Projector: by using the ProjectInfo command or by clicking the Question button in the Check Out or Check In window. The information that you can retrieve from a project includes

- **Project information**
  - □ Author
  - □ Last modification date of the project
  - □ Project comment
- **Revision tree (file) information**
  - □ Author
  - □ Date original file was added to the project
  - □ Last modification date of the revision tree
  - □ Revision tree comment
- **Revision information**
  - □ Author
  - □ Task
  - □ Date the revision was created
  - □ Revision comment

The Check Out window's Info View (see Figure 7-7) is designed to help you browse through the project, finding information about revision trees or individual revisions. The command-line interface can handle more complex batch-type requests, such as "list all revisions, including comments, that Bob made to a particular file, Commands."

The script CompareRevisions lets you compare two revisions side by side, highlighting the differences. Use of this script is described at the end of this section and in Part II.

■ **Figure 7-7** Revision information
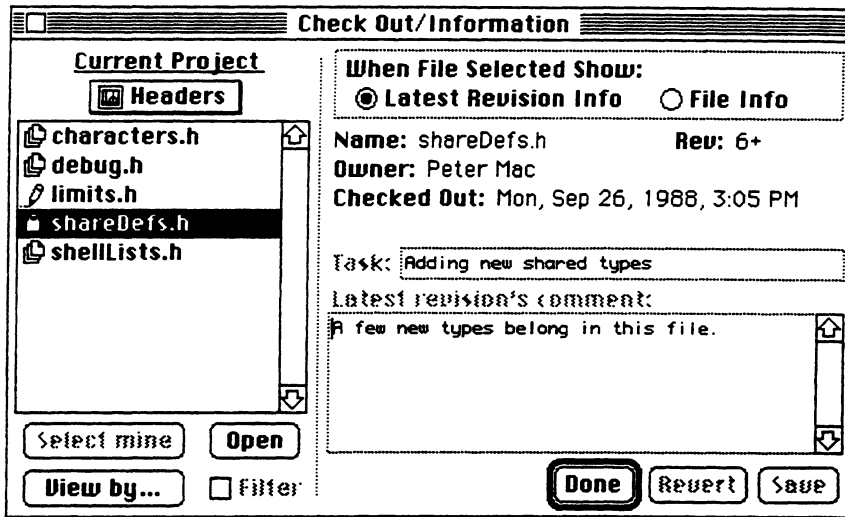
You can also retrieve information in the Check Out window, by selecting a subset of the project to view via the View By dialog (see Figure 7-8).

■ **Figure 7-8** The View By filter
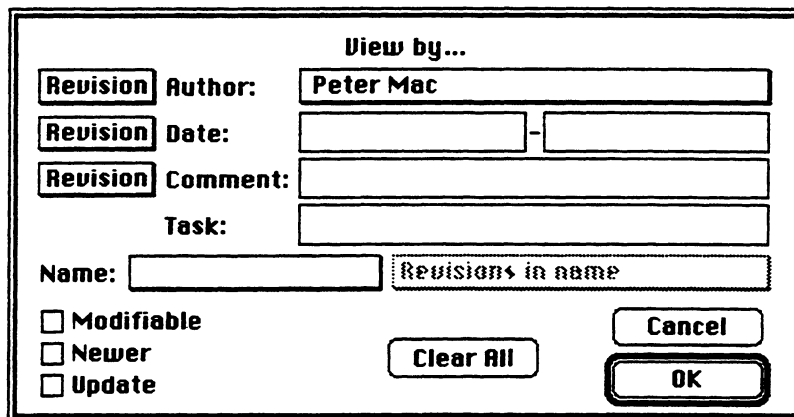
The View By dialog provides different items with which you can filter the revisions in the list. Only revision trees or revisions that match your criteria will be displayed. To specify a filter, bring up the View By dialog box and select the items that are important to you. You may specify these items:

■ The author of a revision tree or revision. All the authors known to the project will be listed in a pop-up menu. Select the desired author from the list.

■ The file modification date or revision creation date. Type in the starting and ending dates. The format is dd/mm/yy [hh:mm[:ss] [AM I PM]]. To specify "on or since a date," enter the starting date in the first box, and leave the second box empty. To specify "before or on a date," enter the ending date in the second box, and leave the first box empty.

■ File or revision comments. Type in either a literal string, or a regular expression in slashes (/regular expression/).

■ Task comments. Type in either a literal string, or a regular expression in slashes (such as: /regular expression/).

■ Name. The pop-up menu contains all your private names followed by the project's public names. Select the desired name from the list. You may also specify a relation to that name (for example, to list all the revisions since alpha). Select the desired relation from the pop-up menu next to the name.

■ Modifiable. List only those revisions checked out for modification.

■ Newer/Update. List only those revisions that would be checked out by using the corresponding option to the CheckOut command.

For the author, date, and comment items, you must specify whether each should be applied to revision trees or to revisions.

◆ Note: The display of all the revision trees is affected unless you specify a "file" filter from the Revision/File pop-up menu.

For example, in Figure 7-9, the user has specified a filter to list all revisions in alpha, created by John Dance, on or after August 12, 1988, dealing with Bug #222.

■ **Figure 7-9**   The "View By" dialog with selection criteria

The following ProjectInfo command is equivalent to the View By dialog in Figure 7-9.

```
ProjectInfo -a 'John Dance' -d '≥8/12/88' -t '/bug=222/' -n alpha
```

Selecting a project displays information about the project (see Figure 7-7). Selecting a revision tree either displays the current state of the revision tree, that is, the status of the latest revision (see Figure 7-7), or it displays the revision tree information. Which is displayed depends on the radio buttons in the upper part of the window's information display (see Figure 7-7). Double-click a filename to display its revision tree. The latest revision is selected by default, and its information (status) is displayed. Selecting another revision displays its status. The Comment and Task fields are editable so that changes or additions can be made to old comments.

Comparing revisions

You can compare two revisions by using the script CompareRevisions. You'll find details on CompareRevisions in Part II.

1. Make sure you have checked out the branch revision against which you want to compare another revision.

2. Execute the CompareRevisions script on the file you have checked out.

Components of a project

This section explains in detail how Projector works. Projector keeps track of these components for each project under its supervision:

- Projects
- Nested projects (subprojects)
- The files (revision trees) belonging to a project
- All revisions of each file (revision tree)
- The branches of each file (revision tree)
- Names of every user creating or modifying a file or project
- Symbolic names

Each of these components is discussed in the sections that follow.

Projects

A **project** consists of a project name, an author, some text describing the project, a set of revision trees belonging to the project, and whatever subprojects the project may have, which are actually projects in their own right. The **author** is the person who created the project.

Projects can reside locally on an individual user's disk, or they can be placed on an AppleTalk file server to facilitate access by multiple users. AppleShare can be used to assign access privileges to various users. (Projector does not itself provide facilities for assigning access privileges.)

Use the Project command to set and show the current project. Projector assumes that all Projector commands pertain to the current project unless told otherwise.

The **project directory** is the directory in which a given project resides. It is defined at the time the project is created. All revisions to all revision trees and all other Projector information are kept in the project directory within the project file, called **ProjectorDB**. All users access the same ProjectorDB. **Nested projects** are also kept in this directory as subdirectories. (See "Nested Projects" later in this chapter.)

Every user has a **checkout directory** for each project. This is the directory into which, by default, Projector places checked-out revisions. You can change the checkout directory by using the CheckOutDir command.

Each user can select one or more projects for access by using the MountProject command. Selecting a project makes it and all its nested projects (subprojects) accessible to the user. You can remove projects from the root project list with the UnMountProject command. The MountProject and CheckOutDir commands allow individuals to customize their own project name space.

It is easy to check out one revision into more than one directory by changing the checkout directory with the CheckOutDir command or by explicitly specifying the directory with the -d option. This makes it easy to look at an old revision of a file or to compare the differences between revisions.

▲ **Warning**   This flexibility of the CheckOutDir command might inadvertently cause problems during check-out because the modifiable revision might not be in the usual checkout directory. ▲

Typically, the UserStartup file, a script, or AddMenu contains a series of MountProject and CheckOutDir commands that connects users to a set of projects. Simply mounting a volume does not give you access to the projects that are contained on that volume. This would be undesirable since many projects may not be of interest to every user.

■ **Figure 7-10**   Sample project check-out configuration
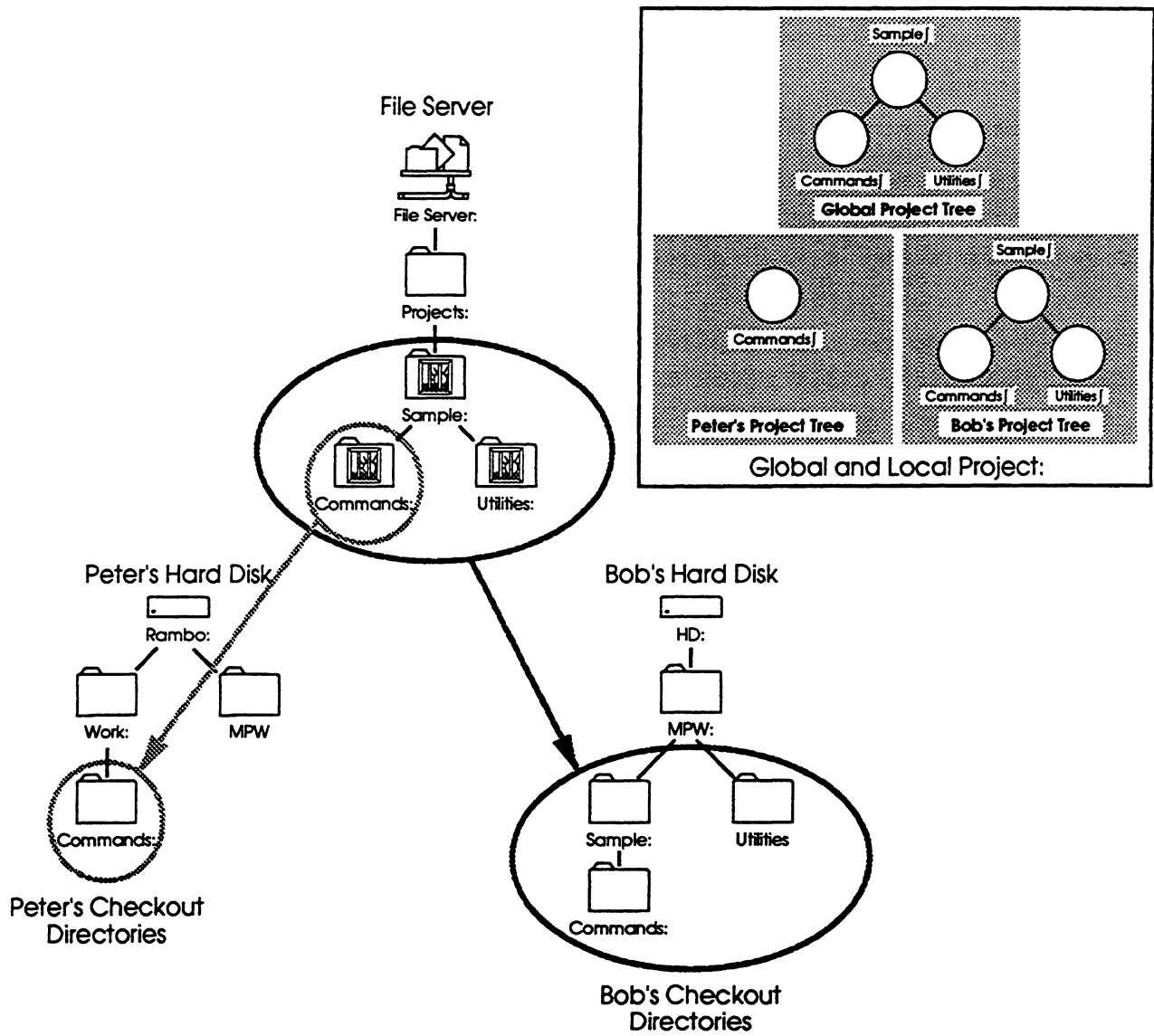
The location of the project directory is the same for everyone, but the checkout directory can be different for each individual. For example, Bob and Peter both access the Commands project, but they have different checkout directories (see Figure 7-10). When Peter checks out files, they go by default to Rambo:work:Commands. Bob's files, on the other hand, go to hd:MPW:Sample Project:Commands:.

Peter's UserStartup could contain the following commands:

```
MountProject FileServer:Projects:Sample:Commands
CheckOutDir -project Commands∫ Rambo:work:Commands
```

Bob's could contain

```
MountProject FileServer:Projects:Sample:
CheckOutDir -project Sample∫ "hd:MPW:Sample Project"
CheckOutDir -project Sample∫Commands "hd:MPW:Sample Project:Commands"
CheckOutDir -project Sample∫Utilities hd:MPW:Utils
```

(See the MountProject and CheckOutDir commands in Part II for more information and examples.)

Projector provides two ways to specify the current project, that is, the project a command will affect. The order of precedence (from greatest to least) is

1. Use the project specified on the command line with the -**project** option.

2. Use the current project specified by the Project command. If you set the current project to the name of a particular project, then you don't need to specify that project with every succeeding use of a Projector command. The current project is thus analogous to the current directory of the Directory command.

If Projector cannot determine the current project, an error is reported and the command is aborted. In the Check Out window you select the current project in the Project list, just as you would select a file in a Standard File dialog box. In the Check In window, click on the "Project:" in the upper-left corner to display the Project pop-up menu.

Nested projects

Projector supports nested projects, also called subprojects. A series of related projects, such as the example projects found in the Examples folder of MPW, can be configured as a hierarchy of projects. People can then access the project structure on any level they choose, much in the same way people use HFS (see Figure 7-10). See Figure 7-11 for a sample project hierarchy.

■ **Figure 7-11**   A sample project hierarchy
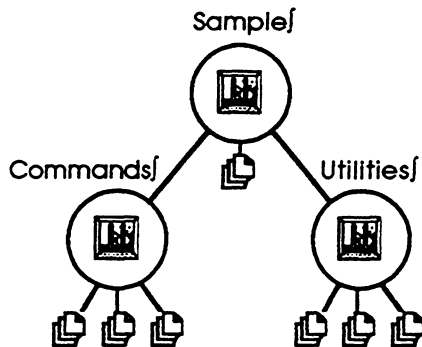
In Figure 7-11 the Sample project is the highest level project, that is, it does not have a parent project. Projects are represented as circles, and revision trees (that is, files with all their associated revisions) are represented as smaller boxes. Just as you can mount several volumes, you can also mount several projects. However, Projector does not allow root projects to have identical names. In Figure 7-11, mounting the Sample project gives you access to all the projects in the project tree.

Projects are named much in the same way as directories, except that the integral character (∫), obtained by pressing Option-B, is used as the name separator. Projector requires full Project pathnames at all times. Partial pathnames are not supported. Use of the integral character at the end of a project path is optional.

■ To avoid confusion with HFS pathnames, Projector does not use colons as project pathname separators. Some commands, NewProject for instance, accept both HFS and project paths as parameters. Because the separators are different, there is no confusion as to what the parameter represents.

■ Integral characters (∫) are not allowed in Project names for the same reason that colons are not allowed in HFS paths.

Revision trees

When a file is added to a project, it forms the first revision of a new revision tree. The new revision tree bears the original file's name. Each revision tree in a project consists of the following components:

■ A name

■ An author, the person who added the original file to the project

■ A comment describing the revision tree

■ A record describing the current state of the revision tree, that is, who has checked out the revision tree and all of its revisions

■ The set of revisions and branches of the original file

Projector can be used with all types of files, such as TEXT, OBJ, APPL, application documents, and so on. The only difference between text files and nontext files is that revisions to nontext files are not compressed, and they cannot be automatically opened at check-out time. Otherwise, there is no distinction between text and nontext files. You can check out read-only copies of nontext files, or check out any such revision for modification and then check it back in as a new revision. Revisions of nontext files can also be named and deleted.

Each time a programmer checks in an updated copy of a revision, a new revision is created. As changes are made and the number of revisions grows, a **revision tree** forms. The revision tree traces the history of the file. By accessing various portions of this tree you can retrieve, inspect, and compare any of the previous revisions of a file. Projector also allows old revisions to be deleted when they are no longer of interest, provided that you delete all old revisions prior to a selected revision. In other words, arbitrary deletion is not allowed.

Once a revision is checked out for modification, it is locked, thus preventing a second modifiable copy of that revision from being checked out at the same time. You may check out a read-only copy of the locked revision.

However, it is possible to check out a modifiable copy of another revision. If you insist on checking out a revision that is already locked, Projector creates a branch for this new copy. You can later merge the changes to synchronize the file. Figure 7-12 shows a revision tree with branches and revisions of branches.

Each revision in a revision tree contains the following identification:

■ A revision number

■ A creation date

■ A Comment field describing the reason for the revision

■ A Task field describing any specific tasks undertaken or any other information

■ The author of the revision

■ A copy of the revision itself (compacted in the case of TEXT files)

■ **Figure 7-12**   A revision tree
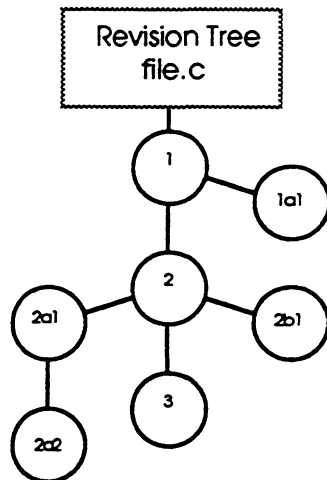
Revisions are normally identified in numeric order, that is, 1, 2, 3, . . . , 99, 100, 101, and so on. However, you can use major/minor numbering instead, that is, 1.1, 1.2, 1.3, . . . 1.99, 1.100, 1.101, . . . 2.1, 2.2, and so on. When a new revision is checked in, Projector will automatically increase its revision number by 1, for example, from 4 to 5, or 4.9.2 to 4.9.3. You can override this action by specifying a different revision number. The only restriction is that this new number must be sequentially greater than the revision that was checked out. Here's the syntax for major/minor numbering.

Revision Numbers: *Major[. Minor...]*

To specify a particular revision in a command, append a comma followed by the desired revision number to the end of the name. In other words, file.c,3 refers to revision 3 of file.c. (Remember that commas are not allowed in project filenames.) The first command in the following example checks out the latest (current) revision of file.c. The second command checks out revision 3 of file.c regardless of what the current revision is

```
CheckOut file.c
CheckOut file.c,3
```

The following command checks in file.c, forcing the revision to 4.1:

```
CheckIn file.c,4.1
```

This command is legal only if the revision that was checked out was less than 4.1—for example, 4, 3.9, 4.0.9, or 2, and so on.

Branches

In addition to supporting a sequence of revisions to a file in a project, Projector also lets you create branches. **Branches** are alternative sequences of revisions that are parallel to the main revision sequence.

In Figure 7-3, Revisions 1, 2, 3, and 4 form the **main trunk** of file.c's revision tree. Revisions that are not on the main trunk form branches. These branches can be easily identified by the alphabetic character embedded in the revision number. For example, you might check out Revision 2 of a file and check it back in as Revision 2a1, instead of Revision 3. This would begin the new sequence, 2a1, 2a2, 2a3, and so on. A second branch off Revision 2 would create Revision 2b1. Revisions off branches follow the same default numbering scheme as revisions on the main trunk, 1, 2, 3, and so on. However, you can use major/minor numbering, with an arbitrary number of minor components.

When specifying a revision, a name such as file.c,2a denotes the latest revision on the "a" branch of Revision 2. If there are two revisions, 2a1 and 2a2, then Revision 2a2 is used.

To refer to particular revisions when using the Check Out window, double-click on a filename to display its revision tree. You can then select and act upon the individual revisions—to check out a particular revision of a file or get information about that revision. To interactively check in a file with a particular revision, click the Revision button in the Check In window. This displays a small dialog that lets you change the number.

User names

Most Projector commands require a user name to keep track of who did what. These commands report an error if no user is specified. Projector reserves the Shell variable {User} as a place to maintain the current user name. The MPW Shell initializes the {User} variable at launch time to the User Name field in the Chooser. When you use Projector interactively, via its windows, the current value of {User} appears in the User field. On the command line there are two ways to specify the current user. In their order of precedence (from greatest to least), they are

1. Use the name given on the command line (via the -u option)

2. Use the name given in the {User} variable.

◆ *Note:* User privileges should be handled by AppleShare. Because AppleShare determines privileges when a network volume is initially mounted, changing the {User} variable will not change the access privileges to those corresponding to the new user.

Symbolic names

Projector supports a general-purpose naming facility, NameRevisions, that allows project users to easily specify revision trees, versions, and branches within a project. Using NameRevisions you can create a name that symbolically represents a set of files almost anywhere Projector accepts a set of files. For instance, you could use a name to specify the "latest revisions of" a set of files that are checked out regularly. You could also use a symbolic name to specify the "alpha version" that labels a set of the revisions that can be used to build an alpha version.

Names are specified by the project user. These restrictions apply:

- The first character of a symbolic name *Name* cannot be a digit (0–9).
- Commas are not allowed anywhere in a name.
- Greater-than or less-than symbols (<, ≤, >, ≥) are not allowed anywhere in a name.
- Dashes (-) are not allowed anywhere in a name.
- Names are not case sensitive.
- Names are kept on a per-project basis and can refer, at most, to one revision per revision tree in that project.

For example, the following commands create the symbolic name Work for three revisions that you always check out together. Thereafter you need only check out Work.

```
NameRevisions Work file.c file.h library.c
CheckOut Work
```

Projector cannot create symbolic names by using the Set command and the existing Shell variable mechanism for these reasons:

- Names can refer to only one revision per file. Shell variables are arbitrary text macros, so this restriction could not be enforced.
- Names are kept on a per-project basis. In Projector the **scope** is the current project. In the Shell, scope is based on nested scripts.
- Names do not need special delimiters ({ and }) to be recognized, unlike Shell variables, which do require delimiters.

By default, names are expanded to the revision level when they are defined, not when they are used. Alternatively, by using the -dynamic option, the names will be expanded to the revision level when they are used, not when they are defined. In the above example, the name Work has expanded to the latest revisions of the three revision trees at the time Work was defined. This means that the revisions that Work implies will never change as new revisions to those revision trees are created. You can also explicitly bind a name to a revision (even while using the -dynamic option) by including the revision number at the time of definition. This example illustrates the differences:

```
NameRevisions Work file.c file.h library.c
```

is equivalent to

```
NameRevisions Work file.c,6 file.h,3.5 library.c,7
```

where the specified revisions are the latest revisions of the respective revision trees at the time Work was defined. The -dynamic option allows you to postpone expanding revisions to the revision level until the name is used. For example,

```
NameRevisions -dynamic Work file.c,4 file.h,3 library.c
```

may be equivalent to

```
NameRevisions -dynamic Work file.c,4 file.h,3 library.c,5
```

at one time, and equivalent to

```
NameRevisions -dynamic Work file.c,4 file.h,3 library.c,21
```

at a later point in time.

Names are recursively expanded until no further expansion can occur or until a comma is found. For example, given the following names:

```
NameRevisions defs.h defs.h,1.1
NameRevisions file.c file.c,2.0
NameRevisions -dynamic Work file.c defs.h,2.1 library.c
```

this CheckOut command,

```
CheckOut Work
```

expands to

```
CheckOut file.c,2.0 defs.h,2.1 library.c
```

Because an explicit revision was specified for defs.h in the definition of Work, the expansion of defs.h to defs.h,1.1 does not occur.

Names are particularly useful when working on a branch. For example, suppose you are designing a new algorithm in file.c and want to implement the algorithm on branch 4a of file.c. By defining this name:

```
NameRevisions -dynamic file.c file.c,4a
```

you can automatically check out and check in the latest revisions on the 4a branch.

The command

```
CheckOut -m file.c
```

checks out a modifiable copy of the latest revision on the 4a branch of file.c. You can override the name simply by specifying a particular revision along with the name.

The command

```
CheckOut file.c,3
```

checks out revision 3 of file.c, regardless of any names. Because an explicit revision was given, no name expansion occurs. A comma with no subsequent revision number denotes the latest revision on the main trunk of the revision tree.

The command

```
CheckOut file.c,
```

checks out the latest revision on the main trunk of file.c. If file.c has not been defined as a name, the comma at the end is unnecessary.

Names can be defined recursively in a project tree. Going back to Figure 7-10 as an example, suppose Bob wanted to freeze the current state of his projects and name the current version Release 1. Then the command

```
NameRevisions -a -r -project Sample "Release 1"
```

would create the name Release 1 in each of the projects. This name would thus expand to the latest revisions when the name was defined. This command is equivalent to the following:

```
NameRevisions -project Sample⌡ "Release 1" -a
NameRevisions -project Sample⌡Commands "Release 1" -a
NameRevisions -project Sample⌡Utilities "Release 1" -a
```

Both public and private (the default) names are supported. Public names are visible to all members of the project. Local names are visible only to the individual who created them. Local names can be declared in the UserStartup file by using the NameRevisions command. Public names are stored with the project itself.

Project administration

The administrative duties for projects under Projector are very simple. Anyone who has write access to the project (under AppleShare) can administer the project. Responsibilities include

- moving, renaming, and deleting projects
- deleting old revisions that are no longer needed
- renaming a file in a project
- deleting files that do not belong in the project

Moving, renaming, and deleting projects

You can move or rename a project by using the Finder or the regular MPW commands. Simply renaming the project directory will rename the project.

No other Finder or MPW operations are allowed on project directories.

There are two points to keep in mind when moving or renaming a project:

- When you move or rename a project, the project hierarchy changes; MountProject commands must be modified to reflect the location of the project.
- It is highly recommended that projects be moved or renamed only when no revisions are checked out for modification, and that after the project has been changed *all* read-only copies be checked out again. This is recommended because Projector puts the project name in the resource forks of revisions during check-out. Once the project is moved or renamed, the information is no longer valid.

You can delete an entire project by deleting the folder containing the project. Use the Finder or the MPW Shell's Delete command.

▲ **Warning**   Once you delete the project, all files and their revisions are lost. ▲

Deleting revisions

You can delete old revisions in a revision tree by using the DeleteRevisions command and specifying the oldest revision that you want to keep. All *prior* revisions are then deleted.

For example, if you specify the revision *filename*,5, then *filename*'s revisions 4, 3, 2, and 1 are deleted, leaving *filename*,5 intact. You cannot, however, arbitrarily delete *filename*,4 while leaving revisions 3, 2, and 1 intact. This restriction prevents confusion of the revision numbering scheme.

You can delete entire branches by naming the branch (for instance, *filename*.c,22a).

▲ **Warning**   Once you delete revisions, there is no way to recover them. ▲

If you accidentally check a file into the wrong project, you can remove it and all its revisions by using the **-file** option of the DeleteRevisions command.

Renaming a file in a project

It is impossible to rename a file in a project. Instead, you must check out the file you wish to rename and check the file back in under a different name by first using the OrphanFiles command.

File organization within a project directory

A project resides in an HFS directory called the **project directory**. The name of this directory is the name of the project. You need not worry about the exact file structure within a project directory.

All information regarding a project, including all revision trees, revisions, comments, and so on, is kept in a single HFS file called ProjectorDB with the type `'MPSP'`.

CKID resource

Projector maintains a 'ckid' (Check ID) resource in the resource fork of all files that belong to a project. This is where Projector identifies each revision tree's filename, project, user, revision number, and so on. The structure of this resource is subject to change by Apple.

Projector icons

As you browse through the project hierarchy in Projector windows, look for the following visual cues that convey revision ownership.

Icons Appearing in the Check In Window

Read-only. The file is a read-only file belonging to the current project.

Modified read-only. The file is a modified read-only file belonging to the current project.

The regular document icon represents a file that does not belong to any project. It is visible only when Show All Files is checked.

The pencil icon means that the HFS file is checked out from the current project for modification by the current user.

The lock icon means that the HFS file is checked out from the current project for modification by another user.

File belongs to a project other than current project. It appears only in the Check In window when Show All Files is checked.

Modifiable file belonging to another project (denoted by the tiny plus sign in the lower-right corner). Appears only in the Check In window when Show All Files is checked.

Corrupt 'ckid' resource. Appears only in the Check In window when Show All Files is checked.

Icons Appearing in the Check Out Window

▣ A project. A similar, but larger icon is used in the Finder to represent the ProjectorDB file.

▯ A Projector revision tree. Appears only in Check Out window.

▢ The regular document icon represents an individual revision currently available. It is visible when an individual revision tree is displayed.

✏ When a project is displayed (so that all its revision trees are listed), the pencil icon means that the latest revision of the main trunk is checked out for modification by the current user. When an individual revision tree within a project is displayed (a list of revisions), the pencil icon means that the particular revision is checked out for modification by the current user.

🔒 When a project is displayed (so that all its revision trees are listed), the lock icon means that the latest revision of the main trunk is checked out for modification by another user. When an individual revision tree within a project is displayed (a list of revisions), the lock icon means that the particular revision is checked out for revision by another user.

Projector command summary

The syntax of the commands used to operate Projector are summarized here for your convenience. Detailed information on each of these commands can be found in Part II.

CheckIn -w I -close I [-u *user*] [-project *project*] [-t *task*] [-cs *comment* I -cf *file*] [-p] [-new I -b] [-m I -delete] [-touch] [-y I -n I -c] (-a I *file...*)

CheckOut -w I -close I [-u *user*] [-project *project*] [-m I -b [-t *task*]] [-cs *comment* I -cf *file*] [-p] [-d *directory*] [-r] [-open] [-y I -n I -c] [-noTouch] [-cancel] (-update I -newer I -a I *file...*)

CheckOutDir [-project *project*] [-m] [-r] [-x I *directory*]

DeleteNames [-u *user*] [-project *project*] [-public] [-r] [*names...* I -a]

DeleteRevisions [-u *user*] [-project *project*] [-file] [-y] *revision...*

ModifyReadOnly *filename*

MountProject [-s] [-pp] [-q] [-r] [*project*]

NameRevisions [-u User] [-project Project] [-public I -b] [-r] [[-only] I name [[-expand] [-s] [-replace] I [(names... [-dynamic]) I [-a]]]]

NewProject -w I -close I [-u *user*] [-cs *comment* I -cf *file*] *project*

OrphanFiles *file...*

Project [-q I *projectName*]

ProjectInfo [-project *project*] [-log] [-comments] [-latest] [-f] [-r] [-s] [-only] [-m] [-af *author*] [-a *author*] [-df *dates*] [-d *dates*] [-cf *pattern*] [-c *pattern*] [-t *pattern*] [-n *name*] [*object...*]

TransferCkid *sourceFile destinationFile*

UnmountProject [-a I *project*]

Chapter 8   The Build Process

THIS CHAPTER DESCRIBES THE MECHANICS OF BUILDING A PROGRAM. The steps involved are nearly the same for applications, desk accessories, stand-alone code resources, drivers, and MPW tools. (However, you'll find special instructions on building your own MPW tools in Chapter 12.) All programmers should read the opening sections of this chapter, which explain the entire build process for an application, the usual case. Later sections explain what's different about building a desk accessory, stand-alone code resource, or driver.

Those new to MPW should first read "Building a Program: An Introduction" in Chapter 2. This brief introduction takes you through the steps of using the Directory and Build menus to build a simple program. For more detailed information on using the linker and how it works, see Chapter 10, "More About Linking." ∎

Contents

Overview: the build process

Building a program consists of the following steps:

1. *Create source files and compile them.* Source files are compiled or assembled to produce object files. For information on writing programs in MPW Pascal, MPW C, or MPW assembly language, and including the proper interface or include files, see the appropriate language manual. Chapter 12 describes writing an MPW tool.

2. *Create noncode resources with ResEdit or Rez.* If your program requires any additional resources (other than code resources), you can create them by using the resource editor (ResEdit) or resource compiler (Rez). These may be decompiled with DeRez into text files that can be further modified by the Shell editor before being recompiled with Rez (see Figure 8-1). See Chapter 11 for detailed information.

3. *Create the final executable file with Link.* The object files are linked together, along with any needed library routines, into either a new resource file or an existing one (replacing the 'CODE', 'DRVR', or other executable resources). The output of Link should be placed in the same file as any resources created in Step 2 (except in the case of drivers, as noted in the next paragraph).

◆ *Note:* To build a desk accessory or driver in Pascal or C, an additional step is required: you must run Rez to create the final 'DRVR' resource. For details, see "Building a Desk Accessory or Driver," later in this chapter.

Figure 8-1 illustrates the complete process.

■ **Figure 8-1**   The Build process


```
              ┌─────────────┐
              │ Shell editor│
              └─────────────┘
                                              ┌─────────┐
                                              │ ResEdit │
                                              └─────────┘

    ┌──────────┐                          ┌──────────┐
    │ Source   │                          │ Resource │
    │ files    │                          │ files    │
    │ .a .p .c │                          │ .rsrc    │
    │ 'TEXT'   │                          │          │
    └──────────┘                          └──────────┘

┌──────────────────────┐              ┌──────────────────────┐
│ Compiler or Assembler│              │ Resource decompiler  │
│                      │              │ DeRez                │
└──────────────────────┘              └──────────────────────┘

 ┌────────┐   ┌──────────┐             ┌──────────┐
 │ Object │   │ Libraries│             │ Rez      │
 │ files  │   │ .o       │             │ Source   │
 │ .o     │   │ 'OBJ '   │             │ files    │
 │ 'OBJ ' │   │          │             │ .r       │
 └────────┘   └──────────┘             │ 'TEXT'   │
                                       └──────────┘

      ┌────────────┐              ┌──────────────────┐
      │   Linker   │              │ Resource compiler│
      └────────────┘              │ Rez              │
                                  └──────────────────┘

  ┌────────┐       ◇ Executable ◇
  │ .code  │         program
  │ 'APPL' │       (code & resources)
  └────────┘         'APPL'
                     'MPST'
```
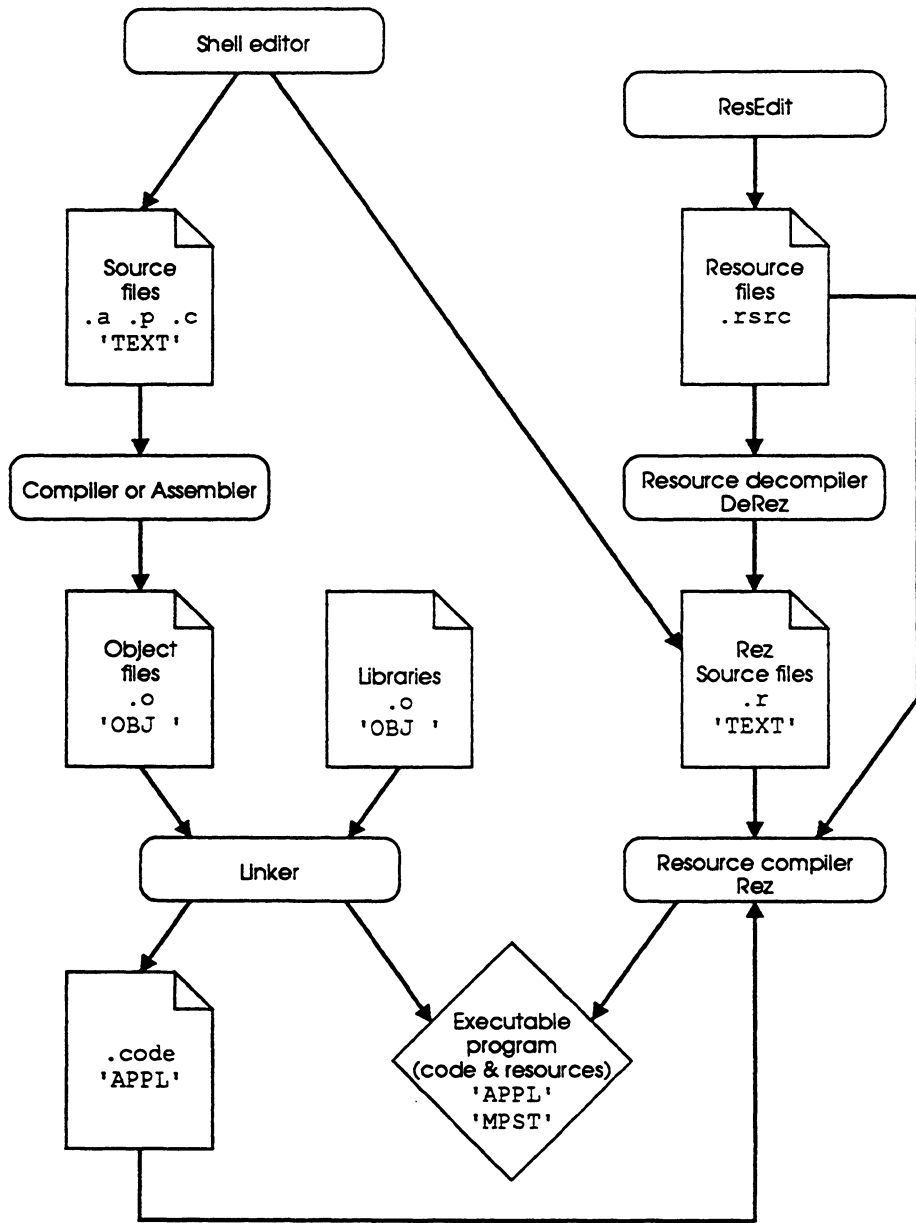
In Figure 8-1 you'll notice that the output from the linker may be placed in a file with the ".code" extension. That file is then reprocessed with Rez to build the final application program. Also keep in mind that it is usually best to run Rez before running Link, even though Rez appears to the right in Figure 8-1. If you do run Rez after running Link, remember to use the **-append** option.

For example, the following series of commands compile, run Rez to compile the resource file, and link the sample Pascal application Sample.p:

```
Pascal      Sample.p
Rez   Sample.r   -o Sample
Link Sample.p.o  ∂
      "{Libraries}"Interface.o  ∂
      "{Libraries}"Runtime.o   ∂
      "{PLibraries}"Paslib.o   ∂
      -o Sample
```

This process is usually automated by using the Make tool. (See the sample makefiles in the Examples folders, and "Using Make" in Chapter 9.)

◆ *Note:* If you build an application with customized icons for documents (that is, a 'BNDL' resource for bundling 'ICN#' and 'FREF' resources), then you need to use SetFile to set your application's bundle bit like this:

```
SetFile -a B MyApp
```

See the chapter "Finder Interface" of *Inside Macintosh* for information.

The structure of a Macintosh application

Macintosh files have two forks: a resource fork and a data fork. The **resource fork** contains a number of resources. The **data fork** may contain anything the application puts there. On the Macintosh, a program is a file whose resource fork contains code resources ('CODE' or other executable resources), and in most cases additional resources containing strings, dialogs, menus, and the like. The code resources for applications and tools must contain a main program (an execution starting point). Desk accessories and drivers, by contrast, don't require a main program, but instead contain collections of routines that are called individually when the desk accessory or driver is used.

The simplest possible application has two resources in the resource fork and nothing in the data fork. The first resource is a 'CODE' resource with ID = 0. (The linker creates this resource, which contains the jump table and information about the application's use of parameter and global space.) The second resource is a 'CODE' resource with ID = 1, which contains the application's code segment. For more information, see the chapter "Segment Loader" of *Inside Macintosh*.

Linking

This section describes how to link an application, desk accessory, or driver. (The process is very similar in the case of MPW tools. Special information on linking MPW tools can be found in Chapter 12.)

For more detailed information about linker functions, see Chapter 10. The Link command itself is described in Part II. The MPW object-file format is described in Appendix H.

Use the command Link to link object files into an application, desk accessory, driver, or other executable resource. By default, linked segments are placed in 'CODE' resources in the resource fork of the output file. Link links together the compiled or assembled object files, along with any needed library routines, into either an existing resource file (replacing the 'CODE', 'DRVR', or other executable resources of the type that it is creating) or a new one (Figure 8-2).

■ **Figure 8-2**   Linking
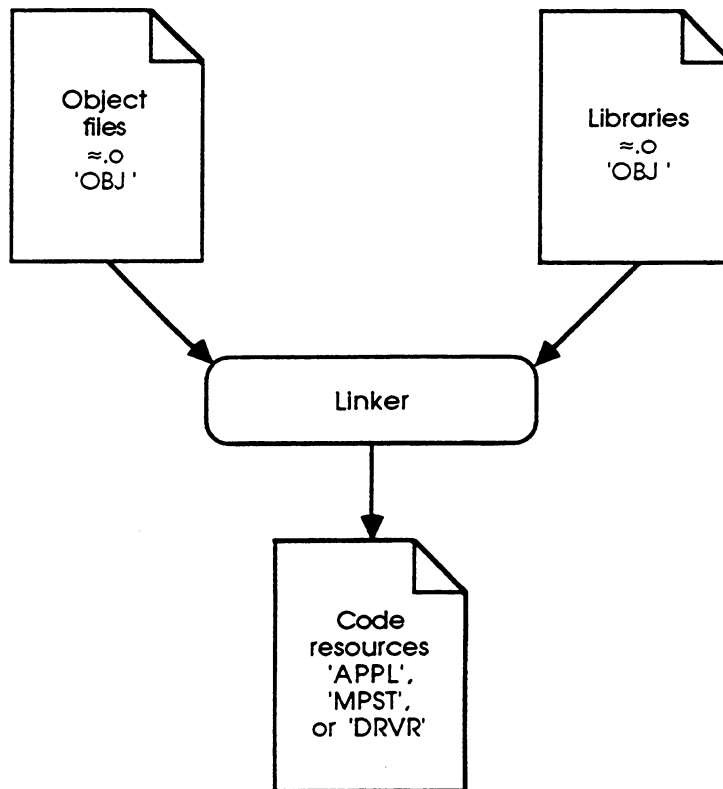
The linker resolves all symbolic references and controls final program segmentation. A related tool, Lib, provides facilities for modifying and combining object files (libraries).

Link's default action is to link an application (type APPL, creator "????"), placing the output segments into 'CODE' resources. You can set a file's type and creator with Link's -t and -c options. (See "File Types and Creators" later in this chapter.)

What to link with

Applications, MPW tools, and desk accessories should be linked with the libraries listed in Table 8-1. It's wise to link new programs with all of the libraries that might be needed. If unnecessary files are specified, Link displays warnings indicating that they can be removed from your build instructions.

■ **Table 8-1**   Files to link

Files	Description
Inside Macintosh interfaces	
{Libraries}Interface.o	
Runtime support	
Link with *one* of the following:	
{Libraries}Runtime.o	If no part of the program is written in C
{CLibraries}CRuntime.o	If any part of the program is written in C
Pascal libraries	
{PLibraries}PasLib.o	Pascal language library
{PLibraries}SANELib.o	SANE numerics library
C libraries	
{CLibraries}CInterface.o	Macintosh interface for C
{CLibraries}CSANELib.o	SANE numerics library
{CLibraries}Math.o	Math functions
{CLibraries}StdCLib.o	Standard C library
Specialized libraries	
{Libraries}ObjLib.o	Object-oriented programming (Pascal and Assembler)
{Libraries}ToolLibs.o	Routines for MPW tools
Desk accessories	
{Libraries}DRVRRuntime.o	Driver runtime library

For details about linking stand-alone code resources or desk accessories refer to "Building a Stand-Alone Code Resource," or "Building a Desk Accessory or Driver," later in this chapter. Details on linking an MPW tool can be found in Chapter 12. MPW tools libraries (the Cursor Control and Error Message File manager routines) are listed in Appendix F. The library of 3-D graphics routines is in Appendix G.

Linking multilingual programs

When you link programs that use libraries from more than one language, the linker may detect several duplicate entry points. Normally it doesn't matter which of the duplicate copies of a particular routine are linked with your program. (You can use Link's -d option to suppress the duplicate definitions warnings.)

However, programs written partly in C and partly in assembly language or Pascal require special precautions. When you link C code with other languages, link with the file CRuntime.o and *not* with Runtime.o. If execution is expected to begin with the C function main ( ) , no special action is necessary. However, if your main program is written in assembly language or Pascal, but part of your program is written in C, then you must do one of two things:

1. Place the object file containing your main program *before* CRuntime.o in the list of object files passed to Link.

2. Use Link's -m option to specify the name of the main routine.

For more hints on using Link's -m option, see "Dead Code" later in this chapter.

File types and creators

When you execute a command, the Shell determines how to run it based on its file type. Files of type APPL are considered applications and are run as if launched from the Finder. Files of type MPST are considered MPW tools and are run within the Shell environment. Files of type TEXT are taken to be scripts and are interpreted by the Shell. An attempt to run a file of any other type produces an error message. Table 8-2 summarizes file types and creators.

■ **Table 8-2**   File types and creators

Type of program	Type	Creator
Application	APPL	*any*
MPW tool	MPST	'MPS'
Desk accessory	DFIL	DMOV
Script	TEXT	*any*

See Table E-1 in Appendix E for a complete list of special MPW file types.

◆ *Note:* Each application has its own unique creator (or **signature**). For more information see the chapter "Finder Interface" of *Inside Macintosh.* For example, creating a file with the type DFIL and creator DMOV tells the Font/DA Mover that this file contains desk accessories.

You can set a file's type and creator with the -t and -c options to Link, Rez, or SetFile.

Building a stand-alone code resource

When developing programs for the Macintosh environment, it is often desirable to build stand-alone code resources. For example, you may want to create custom controls. Some of these resources are

WDEF	window definition procedure (for custom windows)
CDEF	control definition procedure (for custom controls)
LDEF	list definition procedure (for List Manager)
MDEF	menu definition procedure (for custom menus)
INIT	a code resource that is loaded and run at boot time by the system startup code
XCMD	external command for Hypercard

These rules must be observed to create a stand-alone code resource:

1. No global or static variables can be declared. No calls can be made to library routines that use global variables (such as printf( )).

2. If you are using string or floating-point constants in C source, you'll usually need to use the C compiler's -b option to put those constants in the code segment (rather than generating global variables).

3. You must use Link's -rt option to specify the code resource type (such as 'WDEF', 'INIT', and so on) and the resource ID.

4. Because most stand-alone code resources are called as if they were Pascal procedures, you must declare the main procedure with the Pascal keyword in C.

5. You must use Link's -m option to specify the entry point for the code resource if you want dead-code-stripping (see "Dead Code" in the hint section that follows). The procedure that is the main procedure for the stand-alone code resource must be the first procedure in the source file, and that source file's object file must be the first file in Link's list of object files to link. In the case of MPW C, you must make the main entry point the first function in your file (including all #include files) if your main entry point is not named "main" or if it is named "main" but is of type Pascal, as required for a CDEF.

6. If you need to place all of your object modules in one resource (as in the case of a CDEF), use Link's -sn and -sg option to combine several segments into one segment.

◆ Dead code

Given an entry point to a module, the linker loads object files and creates a table of all references reached from that point. The table, called a **Directed Acyclic Graph**, is a tree of all reachable modules. It tracks every single module going into a link. For example, 500 modules may be submitted to linking when only 100 of them will actually be used by the final linked object. The remaining 400 modules cannot be reached by references stemming from the main entry point and are therefore considered **dead code**.

Here's how to use Link's -m option with main modules:

When you link a 'CODE' resource with ID equal to zero (that is, a normal application), you must specify a main module with Link's -m option or specify a module in one of the object files included with the link.

When you link any other type of resource (such as a 'DRVR' or 'CODE' resource with an ID other than zero), the linker doesn't require a main module. However, you can specify a main module by using the -m option or by specifying Main at assembly time. Then, if there is a main module, the linker strips out unreachable modules, that is, dead code. If there is no main module going into a link, then the linker does not remove dead code. In that case, whatever libraries you submit to the linker will be included in their entirety.

In some cases, for instance, code to be used in a ROM, you might not have a single entry point; there might be a number of possible entry points. In such cases you want to be careful to submit only the modules that will actually be needed by your program. ◆

Here is a sample CDEF written in C and Pascal that draws boxes:

```
/* File LinesCDEF.c
   Copyright Apple Computer, Inc. 1988
   All rights reserved.
*/
```

```
/* This file implements a control definition proc for drawing boxes. */

#include     <Types.h>
#include     <QuickDraw.h>
#include     <Controls.h>

#define      CurrentA5    *((long *)0x904)
#define      GrayPat      (**((Pattern **)CurrentA5) - 0x18)

pascal long BoxControl (short              varCode,
                        ControlHandle      theControl,
                        short              message,
                        long               param)
{
        if (message == drawCntl && (*theControl)->contrlVis) {
                FrameRect(&((*theControl)->contrlRect));
        } else if (message == testCntl) {
                return PtInRect(*(Point *)&param,
&(*theControl)->contrlRect) &&
                        ((*theControl)->contrlHilite != 255);
        }
        return 0;
}
```

The makefile rules to build the above sample into a program called Application
look like this:

```
Application ƒƒ    LinesCDEF.c.o
     Link -rt CDEF=258 ∂
          -m BOXCONTROL ∂
          -sn "Main=Lines" ∂
          LinesCDEF.c.o ∂
          -o Application
```

Here is the same sample CDEF code written in Pascal:

```
{  File LinesCDEF.p
   Copyright Apple Computer, Inc. 1988
   All rights reserved.
}

{ This file implements a control definition proc for drawing boxes. }

UNIT LinesCDEF;

INTERFACE
     USES
          Memtypes, QuickDraw, OSIntf, ToolIntf;

IMPLEMENTATION
```

```
FUNCTION BoxControl (varCode:         INTEGER;
                     theControl:      ControlHandle;
                     message:         INTEGER;
                     param:           LONGINT) : LONGINT;
BEGIN
      BoxControl := 0;
      IF (message = drawCntl) AND (theControl^^.contrlVis <> 0) THEN
            FrameRect(theControl^^.contrlRect)
      ELSE IF message = testCntl THEN
            BoxControl := ORD4(PtInRect(Point(param),
            theControl^^.contrlRect) AND
                                    (theControl^^.contrlHilite <> 255));
END;

END.
```

Here are the makefile rules to build the above sample into a program called Application:

```
Application ƒƒ    LinesCDEF.p.o
      Link -rt CDEF=258 ∂
            -m BOXCONTROL ∂
            -sn "Main=Lines" ∂
            LinesCDEF.p.o ∂
            -o Application
```

Building a desk accessory or driver

Putting together a desk accessory or driver in languages other than assembly language requires two steps:

1. Link your driver code with the DRVRRuntime library (this must appear first) and with any other libraries you need. The object code is linked into a code resouce of type 'DRVW', an intermediate form of the standard 'DRVR' resource. (The DRVRRuntime library is the header code for operating desk accessories that cannot be created in C or Pascal. See the section "The DRVRRuntime Library" later in this chapter.)

2. Use the resource compiler, Rez, to create the final driver file. That is, compile the linked 'DRVW' resource into a standard 'DRVR' resource, using the 'DRVW' type declared in :RIncludes:MPWTypes.r, together with any other resources your desk accessory may require.

You then install your desk accessory in the System file by using the Font/DA Mover.

Figure 8-3 illustrates the process of building a desk accessory or other driver.

■ **Figure 8-3**   Building a desk accessory with DRVRRuntime
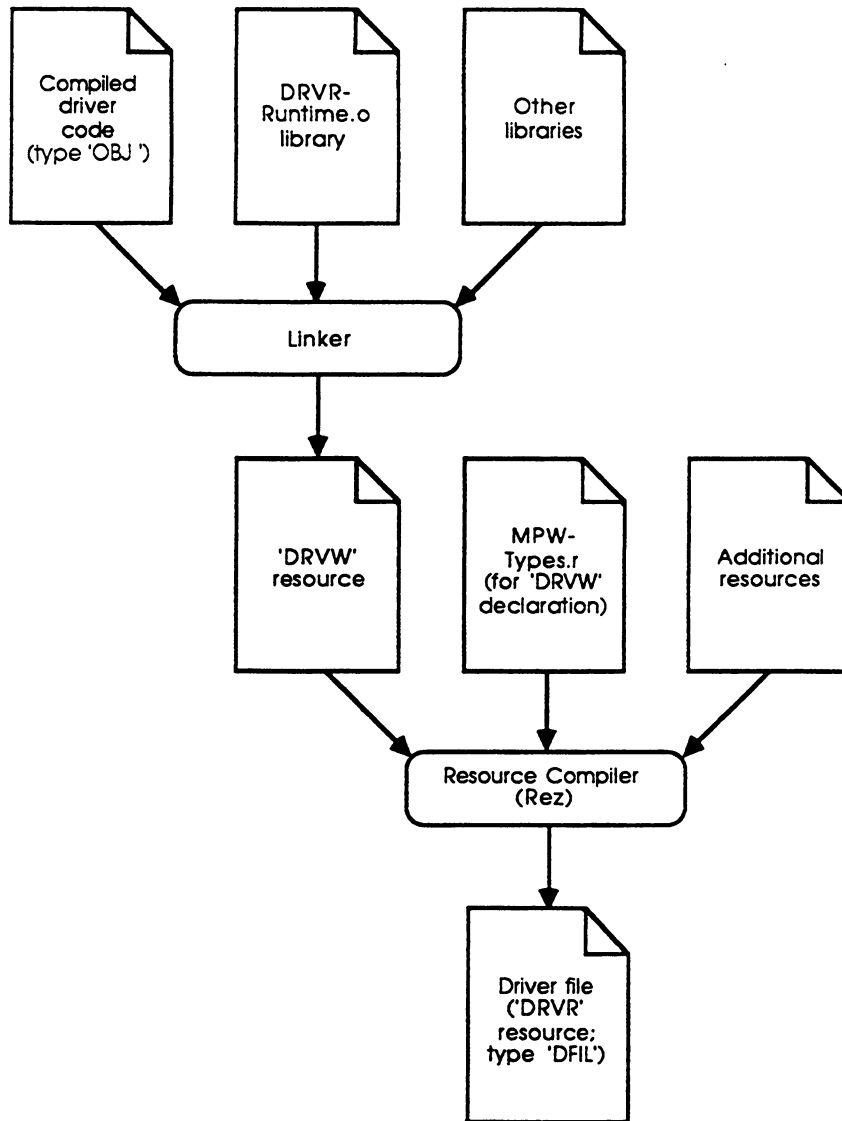
◆ *Note:* Of course, it's always possible to create a desk accessory directly in assembly language, without using DRVRRuntime.

Linking a desk accessory or driver

These rules must be observed to link a driver or desk accessory:

- Link's -rt option must be specified. The -rt option indicates the link of a desk accessory or driver and sets the resource type and ID. (The default, if no -rt option is specified, is to output 'CODE' resources beginning with resource ID 0.)

- The code must be in a single segment (that is, no jump table is constructed). You can map code from several segments into a single segment with the -sg or -sn options.

- Desk accessories written in Pascal or C must be linked with DRVRRuntime.o, which should appear first in the list of object files.

For example, the following command links the sample desk accessory file Memory.c.o, placing the output in the file Memory. (This output is the intermediate 'DRVW' resource, which must be converted into a 'DRVR' resource as explained in the next section.)

```
Link   -rt DRVW=0  ∂
       -sn Main=Memory  ∂
       "{Libraries}"DRVRRuntime.o        # must appear first ∂
       Memory.c.o  ∂
       "{CLibraries}"CRuntime.o  ∂
       "{CLibraries}"CInterface.o  ∂
       -o Memory.DRVW
```

This command has these results:

- The -rt option sets the output resource type to 'DRVW' and the resource ID to 0.

  - ◆ *Note:* This ID must match the ID specified in the $$resource statement in the Rez input file. Note also that any additional resources "owned" by the desk accessory must observe a special numbering convention, as described in the chapter "Resource Manager" of *Inside Macintosh.*

- The -sn option combines the segment Main into the segment Memory.

- The specified files are linked. The DRVRRuntime.o library must be the first object file in the link list. This ordering ensures that the main entry point in CRuntime.o is overridden by the DRVRRuntime.o entry point. (A linker warning will call attention to this requirement unless you suppress it with the -w option.) The main entry point in CRuntime.o cannot be used for desk accessories.

◆ An easy way to make sure all the code is in one segment

As mentioned in the preceding list of instructions, you must keep your code in a single segment when linking a desk accessory or driver. Use the -sg option without an equal (=) sign. For example

```
Link -rt DRVW=0∂
     -sg Memory∂
     "{Libraries}" DRVRRuntime...                                      ◆
```

△ **Important**   Desk accessories must not call routines that use global variables, and therefore are less likely to need routines from the Pascal, C, and specialized libraries listed in Table 8-1. In a correct link, the linker's progress information will report "Size of global data area: 0," and "No data initialization." If global data is somehow allocated, the link will succeed, but the desk accessory will not run correctly. △

The desk accessory resource file

The last step in constructing a desk accessory or driver is to put together the DRVR header with the linked code. The following example of a resource compiler (Rez) input file shows how this is done:

```
#include "Types.r"
#include "MPWTypes.r"
type 'DRVR' as 'DRVW';
#define DriverID 12   /* The number must be in the range [12...26] */
resource 'DRVR' (DriverID, "\0x00Memory", purgeable) {
            dontNeedLock,    /* OK to float around, not saving
                                    ProcPtrs */
            needTime,        /* Yes, give us periodic Control calls */
            dontNeedGoodbye, /* No special requirements */
            noStatusEnable,
            ctlEnable,       /* DA's do only Control calls */
            noWriteEnable,
            noReadEnable,
            5*60,            /* Wake up every 5 seconds */
            updateMask,      /* This DA only handles update events */
            0,               /* This DA has no menu */
            "Memory",        /* This isn't used by the DA */
            $$resource("Memory.DRVW", 'DRVW', 0)
};
```

The header information contains the details of the desk accessory's event mask, menu ID, and so on. (See the chapter "Device Manager" of *Inside Macintosh* and the file MPWTypes.r for information about the format of a `'DRVR'` resource.) The `$$resource` directive then appends the linked object code to the DRVR header where it belongs.

If your desk accessory has any owned resources, such as `'STR#'` or `'WIND'` resources, you can add them to your desk accessory's resource compiler input.

To build the desk accessory resource, use the Rez command to compile the resources you have specified, and set the file type and creator for a Font/DA Mover document:

```
Rez -c DMOV -t DFIL Memory.r -o Memory
```

The file type DFIL indicates a document file for the Font/DA Mover; the creator DMOV indicates a Font/DA Mover document (the suitcase icon).

To install a desk accessory, use the Font/DA Mover to place the desk accessory in the System file. You can do this from MPW as follows:

```
"Font/DA Mover" Memory
```

After exiting the Font/DA Mover, you can execute the desk accessory by selecting its name from the Apple menu.

The DRVRRuntime library

This section documents the DRVRRuntime library and describes the specifics of writing a desk accessory or other driver by using MPW and MPW Pascal or MPW C. If you are using assembly language, you don't need DRVRRuntime. (See *Inside Macintosh* for details on creating a desk accessory.) Because a desk accessory is a special case of a driver, all of the information in this chapter applies to both. You should already be familiar with the following:

- "Writing Your Own Desk Accessories" in the chapter "Desk Manager" of *Inside Macintosh*

- The chapter "Device Manager" of *Inside Macintosh* (for information about `'DRVR'` resources, and so on)

- "Building a Desk Accessory or Driver" earlier in this chapter

For information about the actual routines used in Pascal, C, or assembly language, see the appropriate MPW language reference manual.

A **desk accessory** is a 'DRVR' resource whose resource name begins with a null character ($00). 'DRVR' resources reside in the System file. Desk accessories have traditionally been written in assembly language, partly because of the peculiar 'DRVR' resource format used for desk accessories. Setting up the 'DRVR' layout header, passing register-based procedure parameters, and coping with the nonstandard exit conventions of the driver routines have made it difficult to implement desk accessories in higher level languages. To overcome these difficulties and to simplify the task of writing a desk accessory in Pascal or C, MPW provides the following:

- The library DRVRRuntime.o, which contains the "glue" for setting up your *open, prime, status, control,* and *close* routines.

- The resource type 'DRVW', declared in {RIncludes}MPWTypes.r. The 'DRVW' resource is an intermediate form of the 'DRVR' resource and contains constants that point to the addresses of the driver routines in DRVRRuntime.o.

The DRVRRuntime library contains a main entry point that overrides the main entry point in CRuntime.o or in your Pascal or assembly-language source. The DRVRRuntime entry point contains driver glue that sets up the parameters for you, calls your routines, and performs the special exit procedure required for a desk accessory to return control to the system. Your routines perform the actions of the desk accessory, such as opening a window and responding to mouse clicks in it.

The resource compiler input (resource description file) for your desk accessory includes the details of your desk accessory header, such as its driver flags, event mask, menu ID, and driver name. The driver is built by coercing the intermediate 'DRVW' resource to a resource of type 'DRVR', which is the format required for desk accessories. Your resource description file also specifies resources for any strings, windows, and menus used in your desk accessory. (For an example of such a resource description file, see "The Desk Accessory Resource File" earlier in this chapter.)

Using DRVRRuntime.o has several advantages:

- No assembly-language source code is required.

- The resource compiler is an integral step in the build process, permitting the easy addition of a desk accessory menu or other owned resources.

- The programmer's interface to the *open, prime, status, control,* and *close* routines uses standard calling conventions. Each function returns a result code which is passed back to the system.

- The DRVRRuntime glue handles the proper exit conventions. (Drivers have peculiar exit conventions, requiring immediate calls to exit via an RTS instruction, but non-immediate calls to jump to the IODone routine—these exit procedures cannot be expressed in C or Pascal.)

Together, the DRVRRuntime library and the 'DRVW' resource form the dispatch mechanism to your driver routines. The next section describes the structure of your driver routines.

What your routines need to do

To write a driver, you need to write five functions named DRVROpen, DRVRPrime, DRVRStatus, DRVRControl, and DRVRClose.

◆ *Pascal note:* In Pascal, you'll need to write a unit that declares these five functions in your interface.

Each of these functions is declared to use Pascal calling conventions, so that the DRVRRuntime library is available for use by both C and Pascal programmers. (See the appropriate language reference manual for details.)

The calling sequence for all five driver routines is the same: The parameter ioPB is the pointer to the driver's I/O parameter block (passed from the system in register A0), and dCtl is the pointer to the driver's device control entry (from register A1). The function returns a result code, which DRVRRuntime puts in register D0. This result code is a Pascal integer (C short). Desk accessories always return a result code of 0.

For example, here is the Pascal declaration for your DRVROpen routine:

```
FUNCTION DRVROpen(ctlPB: ParmBlkPtr; dCTl: DCtlPtr): OSErr;
```

Types ParmBlkPtr and DCtlPtr are declared in the file OSIntf.p. Type OSErr is declared in the unit Types.

In C, write the routines like this:

```
pascal OSErr
DRVROpen(ctlPB,dCtl)
        CntrlParam *ctlPB;
        DCtlPtr dCtl;
{
        ...
        return(resultCode);
}
```

Types CntrlParam and DCtlPtr are declared in the file Devices.h. Type OSErr is a short and is defined in Types.h.

Desk accessories only: The body of the desk accessory code resides in your routines
DRVROpen, DRVRControl, and DRVRClose. Your routines DRVRPrime and
DRVRStatus are never called by the system, but the DRVRRuntime library expects them
to be present anyway—they cannot be omitted. It is sufficient to declare them and have
them simply return 0.

Programming hints

In the current release of MPW, global data is not available for use by desk accessories.
That is, variables declared outside your functions cannot be used. In particular, the
following language constructs reference the global data area and *cannot* be used:

Asm	DATA directives
Pascal	UNIT variables
C	static or extern variables

Also note that QuickDraw globals cannot be used directly. Furthermore, you cannot call
library functions that use any of these things. (Look for the linker message "No global data
was allocated.")

Typically, C and Pascal programmers allocate global storage from the heap and use
'STR#' resources for string constants. (In MPW Pascal, constants are allocated as part of
the code module in which they appear. The same effect can be obtained in MPW C by
using the -b option.) If you need to allocate global data from the heap, you can declare a
record containing all of the global variables you need. In your DRVROpen routine, you
should allocate memory from the heap with the size of this record and store its handle in
the dCtlStorage field of the device control entry. Then, to reference an element in the
record, you can use this handle to reference the global variable you want to use.

Sample desk accessory

A sample desk accessory, Memory, is included in the Examples folders for assembly language, C, and Pascal. This desk accessory has the following features:

■ It displays the current amount of free space in both the application heap and the system heap.

■ It displays the number of free bytes on the default volume, along with the name of the default volume.

■ It performs these operations every 5 seconds, so that you can see how your memory conditions change.

Modifying the Build menu and makefiles

The Directory and Build menus are implemented as AddMenu commands and scripts. This lets you customize these menus to serve your own specific needs. In addition, the makefiles created by using the Create Build Commands menu command (script CreateMake) can be modified as your scripts grow in complexity. See Chapter 3 for a description of the Directory and Build menus.

Variables

The Shell variable {Program} is used to remember the name of the most recently built program. It is set by the Create Build Commands menu command and by each of the Build menu commands. {Program} is used as the default program name for the Build menu items.

Scripts

The following scripts implement the Directory and Build menus. Two of these are supported by Commando dialogs, as noted. These scripts are located in the Scripts folder. Each is documented in detail in Part II.

DirectoryMenu	Create the Directory menu
SetDirectory	Set the current directory. (Commando dialog available.)
BuildMenu	Create the Build menu
CreateMake	Create a program makefile. (Commando dialog available.)
BuildProgram	Build the selected program
BuildCommands	Display the commands required to build the selected program

Files

Commands in the Directory and Build menus may create the following files:

<*Program*>.make	Makefile containing build commands for program
<*Program*>.makeout	Build instructions for current build
{MPW}MPW.Errors	Diagnostic output from commands run from menus

UserStartup

The Directory and Build menus are installed by scripts from the UserStartup file. The commands listed below should be in UserStartup. In addition to creating the Directory and Build menus, they create the aliases needed to support the Directory menu.

```
DirectoryMenu  `(Files -d -i "{MPW}"Examples:≈ || Set Status 0) ≥
      Dev:Null``Directory`
BuildMenu
```

The parameters to DirectoryMenu become the initial list of directories in the Directory menu. You can replace or augment the Examples directories with your favorite list of directories.

Modifying the makefiles

As the complexity of your program increases, you can modify the makefile created by the Create Build Commands menu command. You might add new dependencies, specify compiler and linker options, and so on. The Build menu command will continue to build the program, using the modified instructions.

Include dependencies

You may want to modify a makefile created by the Create Build Commands menu command, if you need to overcome the limitations of the CreateMake script. Makefiles created by CreateMake do not include dependencies on `include` files or Pascal USES files. If you plan to change the `include` or USES files, consider modifying the makefile to express these dependencies.

For example, assuming that the C source file Count.c includes the header file Utilities.h, add the following dependency rule to the file Count.make:

```
Count.c.o    ƒ    Utilities.h
```

No build rules are required; just add the dependency rule. Several `include` or USES files can be listed in the same dependency rule, or separate rules can be used for each dependency. Don't forget to specify the directory for files located in another directory.

Library object files

Makefiles created by CreateMake link your program with the selected libraries listed on the CreateMake command page in Part II. The libraries are selected according to the language or languages in which your program is written and according to the program type (application, tool, desk accessory).

If your program calls routines in a library that is not automatically included in the build commands, then modify the makefile to add that library. The library's name should be added in two places: in the dependency rule immediately before the Link command (if you expect to modify the library), and in the list of libraries in the Link command itself.

If you consistently need to add the same library to your makefiles, you can modify the CreateMake script to include it automatically in the dependency and build rules.

Chapter 9 **Make**

THE MAKE TOOL ENABLES YOU TO KEEP TRACK OF ALL OF THE COMPONENTS OF A
PROGRAM and their relationships to each other. When one component of a
program is modified or updated, Make lets you automatically update all other
parts of the program that depend on it. These updates may be such things as
compiles, assemblies, links, and resource compiles.

Make reads a **makefile** that describes the dependencies of the various
components of a program and outputs commands on the basis of those
dependencies. This section describes how to write a makefile and use Make. (The
Make command and command-line options are described in Part II.) ■

Contents

Format of a makefile

A makefile is a text file that describes dependency information for one or more target files. A **target file** is a file to be rebuilt; it depends on one or more **prerequisite files** that must exist or be brought up-to-date before the target can be rebuilt. For example, an application depends on its source file or files, a number of library files, and resource files. If any of a target's prerequisite files are newer than the target, then the target needs to be rebuilt.

A target's prerequisites may themselves be targets with their own prerequisites, and so on. A target that is not a prerequisite of any other target is called a **root**. A makefile may have one or more roots.

A makefile can include dependency rules, variable definitions, and comments. Table 9-1 summarizes the syntax of a makefile, and the sections following the table describe this syntax in more detail.

■ **Table 9-1**   Makefile summary

Syntax rules	Description
targetFile... ƒ [prerequisiteFile...]	Dependency rule, with or without build commands
[*ShellCommands...*]	This rule means that *targetFile* depends upon *prerequisiteFile*. If any of the prerequisites are newer than the target, the subsequent Shell commands are output so that the target can be made up-to-date with respect to its prerequisites.
	Important: Build commands must begin with a space or tab.
targetFile... ƒƒ [prerequisiteFile...]	Dependency rule, requiring its own set of build commands
ShellCommands...	
.[suffix] ƒ .suffix	Default rule (specifies suffix dependencies)
ShellCommands...	
targetDirectory: ... ƒ searchDirectory: ...	
	Directory dependency rule (used with default rules)
variableName = stringValue	Variable definition
# comment	Comment
{name}	Variable reference
"...", '...', ∂	Quotes (as in the Shell)
∂Return	Line continuation character

◆ *Note:* Use Option-F to obtain the ƒ character.

◆ *Note:* Makefile physical input lines may not exceed 255 characters. Logical input lines (made up of one or more physical input lines continued with the continuation character) may be of arbitrary length.

A makefile for the sample Pascal application (Sample) is shown below:

```
###  Variable Definitions  ###
Libs =        "{Libraries}"Interface.o ∂
              "{Libraries}"Runtime.o ∂
              "{PLibraries}"Paslib.o
###  Dependency Rules  ###
Sample        ƒ Sample.r          # Sample depends on Sample.r
   Rez Sample.r -a -o Sample
Sample        ƒ Sample.p.o        # Sample depends on Sample.p.o ∂
              Sample.r            #  and Sample.r
   Link Sample.p.o ∂
       {Libs} ∂
       -o Sample
Sample.p.o  ƒ Sample.p
   Pascal Sample.p
```

Sample makefiles are contained in the Examples folder (introduced in Chapter 2).

Dependency rules

A **dependency rule** specifies the prerequisite files of a given target file, together with a list of the commands needed for building the target file. These commands will be written to standard output if any one of the prerequisite files is newer than the target file or if the target doesn't exist. The general form of a dependency rule is

targetFile... ƒ [*prerequisiteFile...*]
    [*ShellCommands...*]

The first line is called the **dependency line.** It consists of one or more target file names, followed by the ƒ (Option-F) character (meaning "is a function of"), followed by a list of prerequisite files separated by blanks or tabs. Make looks at the modification dates of the prerequisite files (and *their* prerequisites, if any) and decides whether the target needs to be rebuilt.

Because a target's prerequisites may themselves be targets with their own prerequisites, the investigation of prerequisites is recursive and "bottom up." Thus, commands to rebuild lower-level targets are issued, if necessary, before the dependency rule determines whether higher-level targets need to be rebuilt.

All subsequent lines *that begin with a space or tab* are **build command lines.** These are Shell commands that will be output if the target needs updating. (When Make writes these command lines to standard output, the initial space or tab is omitted.) If the dependency rule omits the build commands, the rule expresses only the target's dependencies. The build lines for the target are assumed to be supplied by another rule.

For example,

```
Sample.p.o   ƒ   Sample.p
    Pascal Sample.p
```

The first line in the example is a dependency rule for the Pascal object file Sample.p.o. This rule states that Sample.p.o depends on the source file Sample.p.

The second line is the associated build command line. If Sample.p is newer than Sample.p.o, or if Sample.p.o doesn't exist, the command `Pascal Sample.p` is written to standard output.

The target is built according to that set of build rules whenever it is out-of-date with respect to any of its prerequisites. If no build commands are specified for a dependency line, the build commands are taken from one of the target's other dependency rules, or from default rules if no build rules were specified for the target.

If you specify more than one single-ƒ dependency line for a target, then the target depends on all the prerequisite names on all the lines. However, only one sequence of build commands may be specified for each target.

More than one target filename can appear on the left-hand side of an "ƒ rule." Each target file on the left-hand side depends on all of the files listed on the right side (and has the same build commands, if specified). If more than one target file is specified, it's exactly as if a separate dependency rule had been given for each target. The built-in Make variable {Targ} has the value of the current target.

◆ *Note:* Typically, you'll have more than one target on the left side of an ƒ rule only when expressing dependencies, so you won't include any build rules. If you do supply build rules, you must write them in a generic fashion by using the {Targ} variable because each target is built independently. Contrary to what the syntax might suggest, multiple targets on the left side of an ƒ rule do not imply that Make builds all targets by a single application of the build rules. Therefore you cannot use this construct to express dependencies in which a tool has more than one output file.

You can also write a dependency rule with an **abstract target**, that is, a target that is not actually built but represents a collection of items. A rule with an abstract target has no build rules, just dependencies; the target on the left side of the ƒ rule does not exist. It serves merely to trigger the dependencies for the prerequisite files on the right side of the ƒ rule. Thus, if your makefile represents several different roots that could be built, they can be collected into a single abstract target that (when it is built) triggers the builds for all the separate objects. That is,

```
All ƒ A B C
```

Double-*f* dependency rules

Double-*f* dependency rules are slightly different from the standard single-*f* rules. Syntactically, a double-*f* dependency rule is the same as a single-*f* rule, except that *ff* is used in place of *f*. The difference in use is that more than one double-*f* rule is expressed for an individual target and that each double-*f* rule requires its own set of build commands. Here is a simple example:

```
TargetFile  ff  A  B  D
     build commands-1

TargetFile  ff  C  D
     build commands-2
```

If the target is out-of-date with respect to one or more dependency sets, each of the corresponding sets of build commands will be output (in the order they appear in the makefile). That is, if TargetFile is out-of-date with respect to both A *and* C, then both sets of build commands are output. (In single-*f* rules, only one set of build commands can be specified for any one target.)

If TargetFile is out-of-date only with respect to B, then only the first set of build commands is output. If TargetFile is out-of-date with respect to D, then both sets of build commands are output, because D appears in the dependency sets for both. In other words, the same file can appear as a prerequisite in more than one double-*f* dependency set.

Here is a more realistic example showing how double-*f* rules are typically used to separately control the building of different components of the same file:

```
App  ff  foo.c.o  bar.c.o
     link  foo.c.o  bar.c.o . . . -o App
App  ff  App.r
     rez  App.r . . . -a  -o App
```

Use double-*f* rules only when you have more than one action to take in building a file, and when you want the actions to be independent (that is, triggered by different dependencies and not always occurring together). Double-*f* rules are useful for separately building code and resources, as shown in the makefile for Sample. (For more examples, see the sample makefile at the end of this section.)

The build commands may be left off a double-*f* rule if they are to be supplied by default rules. If build commands are left out of more than one double-*f* rule for the same target, Make applies the default rules only to the first empty set.

Default rules

Default rules express dependencies between pairs of files whose names are the same but whose suffixes differ. They have the following form:

.[suffix1] ƒ *.suffix2*
      ShellCommands...

◆ *Note:* The period must be present directly in front of the suffix for a default rule to be recognized. The period is taken as part of the suffix.

Default rules are powerful because many specific dependencies and build commands can be expressed by a single rule, thereby eliminating the need to specify many similar dependency rules. Make has built-in default rules for assemblies and for C and Pascal compiles. You need to specify only the dependencies not covered by default rules.

For example, in its simplest form a default rule for C compiles might be

.c.o ƒ *.c*
      C {default}.c

In this example the built-in Make variable {default} represents the common part of the filenames matching the rule. The C compiler will be run on the source file with a ".c" suffix and will produce an object file with a "c.o" suffix.

Default rules are applied only when no build commands have been given for a particular target. You can augment the default rules for a particular file by using additional dependency rules, so long as these dependency rules do not include build commands.

If you are planning to have an object file built by a default rule, there is no need to express the dependency on the source file because it is implied by the default rule. Only additional dependencies, such as includes, need to be expressed explicitly.

Make applies default rules only if the file implied by the right-side suffix of the rule exists, or if Make can arrive at a file that exists by further applications of default rules.

If the left side of a default rule has more than one period (or component), there is the possibility that more than one default rule applies. For example, you may have a default rule for building ".o" files and another for building ".c.o" files. Because Make tries to apply default rules by matching the longest suffix first, the ".c.o" rule is tried first.

Default rules of the form

.   ƒ   *.suffix*

specify dependencies between files with any name and files with the same name followed by the given suffix.

◆ *Note:* Default rules of this form slow down Make processing, because the empty left side of the rule causes it to match against all filenames.

Built-in default rules

A compiled or assembled object file is dependent on its source file. This dependency is typically handled by the built-in default rules.

Additional object file dependencies may result from other units that you use or refer to in your source file—these may be `include` files, C header files, or Pascal USES files. These additional dependencies can be expressed by dependency rules with no build line component, leaving the build lines and object-to-source dependency implied by the default rules.

The data fork of the Make tool contains the following built-in default rules:

```
.a.o     ƒ  .a
   {Asm} {AOptions} {DepDir}{Default}.a   -o {TargDir}{Default}.a.o
.c.o     ƒ  .c
   {C} {COptions} {DepDir}{Default}.c   -o {TargDir}{Default}.c.o
.p.o     ƒ  .p
   {Pascal} {POptions} {DepDir}{Default}.p   -o {TargDir}{Default}.p.o
```

{Asm}, {Pascal}, and {C} are built-in Make variables. Their initial values are

{Asm}	Asm
{Pascal}	Pascal
{C}	C

{AOptions}, {POptions}, and {COptions} are initially null; you can customize the built-in default-rule build commands by defining these variables in your makefile. For instance, you might want to specify the location of your Pascal Include files by adding an -i *pathname* option to the default rules by a variable definition of the form

POptions= -i *pathname*

Or you may want to indicate the use of a different C compiler by changing the value of the {C} variable.

You can redefine the {Asm}, {Pascal}, {C}, {AOptions}, {POptions}, and {COptions} variables. Variable definitions can be overridden in your makefile, on the command line (with Make's -d option), or by an exported Shell variable. See "Variables in Makefiles" later in this chapter.

If you cannot sufficiently customize the default rules by assigning to these built-in variables, you can override any of the default rules by placing your own versions of the default rules in your makefile.

{Default} is another built-in variable; its value is the common part of the filenames matched by a default rule (defined dynamically when Make applies the default rule). The {Default} variable is what allows you to write a generic default rule without referring to a specific filename. Because its value is set dynamically by Make, its value cannot be overriden in your makefile.

{DepDir} and {TargDir} are built-in Make variables that allow default rules to work with the target and prerequisite files in different (or the same) directories:

{DepDir}   The directory component of the prerequisite name
{TargDir}   The directory component of the target name

◆ *Note:* {DepDir} and {TargDir} have values only when used in the build commands of default rules for which directory dependency rules were applied. In all other cases these variables evaluate to the null string so that they won't interfere with the normal behavior of default rules. Directory dependency rules are explained in the section that follows.

Directory dependency rules

Normally, default rules work only within a single directory; that is, the target and prerequisite files will have the same directory component because the default rules change only the suffixes of the filenames. Directory dependency rules allow default rules to be applied across directories. Just as default rules imply changing a filename suffix between a target filename and a prerequisite filename, directory dependency rules imply changing the directory prefix of the filenames. Directory dependency rules have the form

targetDirectory : ... ƒ *searchDirectory* : ...

Directory dependency rules are identified by dependency names that end in colons (that is, directory names). For example,

```
ObjFiles:   ƒ   SrcFiles:
```

The above rule, together with the standard default rules, would mean, for example, that ObjFiles:*name*.c.o depends on SrcFiles:*name*.c. See the working example at the end of this chapter.

No build commands may be given for a directory dependency rule. More than one directory name may appear on either side of the rule. The current directory can be specified by a single colon (:) on either side of a directory dependency rule.

Directory dependency rules are applied only during the processing of default rules. If Make is applying a default rule and encounters a target name with a directory component, Make checks for a directory dependency rule for that directory. If one exists, Make tries prerequisite filenames with the directory prefixes given on the right side of the rule. The names are tried in the order they appear in the rule; thus more than one directory name on the right side of a directory dependency rule constitutes a list of directories to search.

◆ *Note:* If default rules are meant to be applied from a directory A: to a directory B: and also within A: (that is, from A: to A:), then A: should appear on both the left and right sides of the directory dependency rule. For example,

```
A:   ƒ   A:   B:
```

Variables in makefiles

You can use exported Shell variables and built-in Make variables within makefiles. You can also define variables within a makefile or on the Make command line. MPW Shell variables are described in Chapter 5.

Shell variables

Make automatically defines exported Shell variables before it reads the makefile, so you can use Shell variables in dependency lines and build commands.

Shell variables in dependency lines are expanded because they are typically filenames or parts of a file. Shell variables in build rules pass through unexpanded so that the Shell will be able to process and expand them.

If Make doesn't recognize a variable reference in a build command line, the build line is left unchanged when it is output so that it can be processed later by the Shell. (Unidentified variables in dependency lines are reported as errors.)

▲ **Warning**   Exported Shell variables override Make variables with the same names. An attempt to redefine a Shell variable in the makefile results in a warning message. ▲

Defining variables within a makefile

Variable definitions are makefile entries of the form

variableName = *stringValue*

Subsequent appearances of {*variableName*} are replaced by *stringValue*. Any leading or trailing blanks or tabs are removed from the variable definition. You can use line continuations to make a *stringValue* arbitrarily long.

When a *stringValue* is continued across lines, a single blank replaces any comments, blanks, or tabs at the end of the continuation line and at the beginning of the line after the continuation. Thus, variable values can conveniently contain lists of files. Note that variable values may contain references to other variables.

One common use of variables is to provide parameters to the directory portion of filenames so that you can easily adapt a makefile to different directory setups. Another use is to create a list of filenames that will be used in more than one place.

◆ *Note:* Make variables are not expanded until they are used in dependency lines or until generated in a build line. Thus, you must define any variables appearing in dependency lines somewhere previously in the makefile because variables in dependency lines are expanded immediately to produce filenames. You can define variables in build lines anywhere in the makefile because variables in build lines are not expanded until after the build lines are generated (that is, after the entire makefile has been read).

You can define a variable on the command line with the Make option -d; this option overrides any definition of the variable within the makefile, thus allowing the definition in the makefile to function as a default.

◆ *Note:* Values of Make variables may not contain the ASCII characters 0 or 1.

Built-in Make variables

The following built-in Make variables have values that are dynamically assigned (and that cannot be overridden) as Make generates the build commands:

{Targ}   The complete filename of the target on the left side of the dependency rule whose build commands are being processed.

{NewerDeps}   A list of the names of all of the target's direct prerequisites that were newer than the target; that is, the files that caused the target to be out-of-date.

These built-in variables can be used only in build command lines because they have no value when dependency lines are processed. They cannot be overridden.

When default rules are applied, the following variables are also defined:

{Default}   The common part of the filenames matched by a default rule
{TargDir}   The directory component of the target name
{DepDir}   The directory component of the prerequisite name

◆ *Note:* When expanding the built-in variables {Targ}, {NewerDeps}, {TargDir}, {DepDir}, and {Default} in build commands, Make automatically quotes their values, if necessary, because they will represent filenames or parts of filenames. Don't quote them yourself.

Quoting in makefiles

The Make command supports several of the Shell's quoting conventions. Quoted items can appear in dependency lines, variable definition lines, and build command lines. The following quotation characters are used:

∂   Quotes the subsequent character; that is, the ∂ is removed and the subsequent character is taken to be a literal character (except when ∂Return is used at the end of a line as a continuation character).

'...'   Quotes the enclosed string. The single quotation marks are removed.

"..."   Quotes the enclosed string, but {...} variable references are expanded, and the escape character ∂ is processed. The double quotation marks are removed.

Quotation characters are processed as follows:

- In dependency lines and in the name part of variable definitions, quotation literalizes the quoted characters (useful for expanding file or variable names).

- On the right side of variable definitions, quoted items are passed through "as is," so that the quoting will take effect when the variable is expanded.

- In build command lines, quoted items are passed through as is, so that the quoting will take effect when the build commands are executed by the Shell.

Line continuation character

Like Shell commands, dependency and variable definition lines can be continued over more than one line by using ∂Return. ∂Return causes the ∂, any blanks preceding the ∂, the return, and any leading blanks on the next line to be replaced with a single space. Comments at the ends of such continued lines do not comment out the continuation character.

Comments in makefiles

The number sign (#) indicates a comment. Everything from the # to the end of the line is ignored. Comments always end at the next return, even if the return is preceded by a ∂.

Comments may appear in dependency lines, variable definitions, and build command lines, or on lines by themselves. Comments in build command lines are passed through to standard output where they are processed as comments by the Shell.

Executing Make's output

Make generates a set of commands, which must be executed separately to perform the actual updates. You can automatically execute Make's command output by calling Make from a Shell script. The simplest form of such a script consists of the two commands

```
Make {"Parameters"} > MakeOut
MakeOut
```

The first command executes Make, using the parameters passed to the script. (See the description of the {"Parameters"} variable in Chapter 5 under "Variables.") Output (that is, build commands) is redirected to the file MakeOut. The second line of the script executes MakeOut.

The order in which Make builds targets

Make builds the top-level target and its prerequisite subtargets in a recursive, "bottom up" fashion. The top-level target (or targets) may be specified on the Make command line. If no target is specified on the command line, then Make builds the first target appearing on the left side of a dependency rule in the makefile (that is, the default top-level target).

The prerequisites of the top-level target (and subtargets) are also investigated in a recursive, "bottom up" order, starting with the first prerequisite in the target's prerequisite list. After the first prerequisite (and its own prerequisites) have been investigated, the target's next prerequisite is investigated. The next prerequisite will be the next one mentioned in the current dependency rule or in the next dependency rule in the file that has the same left-side target.

Thus, the important orderings within a makefile are: the first target mentioned (the default top-level target) and the order of prerequisites for any given target. Otherwise, the order in which targets are mentioned is not important.

Please note, however, that once a target has been investigated by Make it is not revisited, even if it appears somewhere else in the top-level target's prerequisite dependency hierarchy. In other words, while a file may appear as a prerequisite of a number of program components, Make will rebuild it only once (if necessary) when it is first encountered in the recursive "bottom up" traversal of the dependency hierarchy.

Remember that a makefile may have one or more top-level targets (or roots), that is, it may describe how to build more than one object. (The -r option will identify all the roots.) Running Make will rebuild only the targets you specify on the command line. If no targets are specified, Make will rebuild the default targets.

Debugging makefiles

When Make doesn't seem to be doing what you expect, the next step is to debug your makefile. The following procedures are helpful in debugging makefiles:

1. Use Make's **-v** option to generate verbose diagnostic output. This output tells you which files don't exist, which files are up-to-date, which files need rebuilding, and why they are out-of-date. It also points out which files don't have build rules and are thus assumed to be artificial targets. (Targets that are abstract and not really built. See, for example, Note 8 in the Make example that follows this section.)

2. Use Make's **-s** option to show the structure of your target's dependency relations. This option displays the complete structure of dependencies, including those generated by default rules. A target (or subtarget) that doesn't appear or that has no prerequisites may indicate a typographical error in the dependency line for that target (or in the line for one of the targets that depend on it). A target that appears at the wrong level in the dependency graph indicates an error in your dependency specification.

3. Use the **-u** option to find unreachable targets. These may be parts of your target dependencies that did not get connected to the rest of the dependency hierarchy because of a bad or mistyped rule.

Problems due to command generation before execution

Make generates commands that must be separately executed to perform the actual updates. Because Make must determine what build commands to generate before any targets are actually built, the possibility of "phase errors" exists; that is, unexpected behavior may occur when generated commands alter the assumptions that Make used to determine whether targets were out-of-date. (You won't experience these problems unless you have build commands that do things such as deleting files that Make thinks are already up-to-date.)

Problems with different specifications for the same file

You'll experience problems with Make if you use different pathname specifications for the same file (that is, pathnames with different degrees of volume and directory qualification). Make uses the name strings exactly as encountered in dependency lines, so different name strings will result in different entries. (This is done for the sake of performance—no calls are made to the file system, except to inquire about the date of targets that are supposed to be built.) If there is more than one name specification for the same file, each name results in a different Make target, and the resulting dependency relations will be wrong.

Problems with default rules

An error message may appear saying that no rules were available to build something that should have been covered by a default rule. This situation may result from any one of the following problems:

- The default rule may not have matched against anything, and was thus not applied; for example, the default rule

  `.p.o ƒ .p`

  cannot be applied if the .p file does not exist either in the file system or in the makefile dependency specification.

- There may be a typographical error in the filename, so that the default rule could not be applied. You should be able to detect such errors by inspecting the output of Make's -s and -v options.

- There may be a typographical error in a default rule that was given in the makefile, in which case you may not see any dependencies generated by the rule when you use the -s option on the Make command line.

An example

This section lists the makefile used to build an experimental version of the Make tool itself (represented in this makefile by the MakeX target). A series of explanatory notes follows the listing. These notes describe in detail a number of the Make features that were used.

```
################### Variables ####################
ToolDir      =    {Boot}ToolUnits:                            See note 1
ObjDir       =    :Obj:
MakeUses     =    {ToolDir}MacInterfaces.p.o    ∂             See note 2
                  {ToolDir}MemMgr.p.o           ∂
                  {ToolDir}SymMgr.p.o           ∂
                  {ToolDir}Utilities.p.o        ∂
                  {ToolDir}IOInterfaces.p.o     ∂
                  {ToolDir}CursorCtl.p.o        ∂
                  {ToolDir}ErrMgr.p.o           ∂
                  {PInterfaces}IntEnv.p         ∂
                  {PInterfaces}MemTypes.p       ∂
                  {PInterfaces}QuickDraw.p      ∂
                  {PInterfaces}OSIntf.p
MakeObjs     =    {ObjDir}Make.p.o              ∂
                  {ToolDir}Stubs.a.o            ∂
                  {ToolDir}CallProc.a.o         ∂
                  {ToolDir}Utilities.p.o        ∂
                  {ToolDir}Utilities.a.o        ∂
                  {ToolDir}IOInterfaces.p.o     ∂
                  {ToolDir}IOInterfaces.a.o     ∂
                  {ToolDir}MemMgr.p.o           ∂
                  {ToolDir}MemMgr.a.o           ∂
                  {ToolDir}SymMgr.p.o           ∂
                  {ToolDir}SymMgr.a.o           ∂
                  {ToolDir}CursorCtl.p.o        ∂
                  {ToolDir}CursorCtl.a.o        ∂
                  {ToolDir}ErrMgr.p.o           ∂
                  {ToolDir}MacInt.a.o           ∂
                  {ToolDir}MacInterfaces.p.o
Libs         =    {Libraries}Runtime.o         ∂
                  {PLibraries}PasLib.o         ∂
                  {Libraries}Interface.o

LinkOpts     =    -w    # no warnings (for duplicates due to Stubs.a.o)
                                                             See note 3

SourceFiles  =    Make.p                        ∂
                  DefaultRules                  ∂
                  Makefile

########### Default Rule Customizations ############

POptions     =    -i {Boot}ToolUnits:                        See note 4
{ObjDir}      ƒ     :    # directory dependency rule          See note 5
```

```
############### Dependency Rules #################
MakeX                     ƒƒ      {MakeObjs} {Libs}                         See note 6
       Link {LinkOpts} -p -b -o MakeX                    ∂
       -t MPST -c "MPS "                ∂
       {MakeObjs} {Libs} ≥LinkMsgs

MakeX                     ƒƒ      defaultRules
       Duplicate -d defaultRules MakeX -y      # copy default rules into Make's data fork

MakeX                     ƒƒ      Make.r
       Rez Make.r -o MakeX -a                  # Make's Commando resource

MakeX                     ƒƒ      {MakeObjs} {Libs} defaultRules
       SetFile MakeX -m . -d .          #set last-mod and creator dates

{ObjDir}Make.p.o          ƒƒ      Make.p                            See note 7
       Save Make.p ≥Dev:Null || Set Status 0  #save source before compile if changed

{ObjDir}Make.p.o          ƒƒ      {MakeUses}#will be augmented by default rules
                                                                   See note 8


{ToolDir}MacInterfaces.p.o      ƒ              {PInterfaces}MemTypes.p      ∂
                          See note 9
                                     {PInterfaces}QuickDraw.p        ∂
                                     {PInterfaces}OSIntf.p           ∂
                                     {PInterfaces}ToolIntf.p         ∂
                                     {PInterfaces}PasLibIntf.p

{ToolDir}MemMgr.p.o             ƒ      {ToolDir}Utilities.p.o           ∂
                                       {ToolDir}MacInterfaces.p.o   ∂
                                       {PInterfaces}MemTypes.p

{ToolDir}SymMgr.p.o             ƒ      {ToolDir}MemMgr.p.o              ∂
                                       {PInterfaces}MemTypes.p

{ToolDir}Utilities.p.o          ƒ      {PInterfaces}MemTypes.p

{ToolDir}IOInterfaces.p.o       ƒ      {ToolDir}Utilities.p.o           ∂
                                       {ToolDir}MacInterfaces.p.o   ∂
                                       {PInterfaces}MemTypes.p

Backup                          ƒ                                  See note 10
       Duplicate -y ≈ MakeSrc:      #backup to Sony

Restore                         ƒ
       Duplicate -y MakeSrc:≈ :      #restore from Sony

Listings                        ƒ      {SourceFiles}               See note 11
       Print -h -r -ls .85 -s 8 -b -hf helvetica -hs 12 {NewerDeps}
       Echo "Last listings made `Date`" >Listings
```

Notes on Make's makefile

These notes are referenced in the preceding example.

1. The exported Shell variable {Boot}, used in the definition of {ToolDir}, is automatically defined by Make when invoked.

2. Several variables—{MakeUses}, {MakeObjs}, {Libs}, and {SourceFiles}—are used for lists of filenames. This is a convenience because the lists are used in several places later in the makefile; it also helps to reduce errors. Note that you can temporarily remove any file from the list by placing a comment character at the beginning of the line for the file.

3. The {LinkOpts} variable is used to specify linker options (and is used only once). This usage is handy because the definition in the makefile functions as a default that can be overridden from the command line with the -d option, as in

    ```
    Make -d LinkOpts='-w -l >Map'
    ```

4. This directory dependency rule allows the MakeX tool's objects and sources to be in different directories and yet be built by the built-in default rules. In particular, Make.p.o will be in the :Obj: directory while Make.p is in the current directory. Note that for this device to work, Make.p.o must appear with the object directory prefix. Thus it appears in the makefile as {ObjDir}Make.p.o.

5. The {POptions} definition gives a value to one of the variables used in the default rules, customizing the built-in default rules for Pascal compiles for this particular makefile.

6. The four sets of *ff* rules for MakeX (an experimental version of the Make tool) handle (a) the Make link (which creates MakeX's code resources), (b) the copying of the default rules to MakeX's data fork (Make reads the built-in default rules from its own data fork), and (c) the setting of the creation and modification dates. The link will take place only if the MakeX objects or libraries change. The default rules will be copied only if the rules have changed. The resource compiler will rebuild Make's Commando resource only if Make.r has changed. And the setting of the dates will take place if either of the first two rules was activated. (Note that the fourth rule has the union of the dependency relations of the first two.)

7. The two sets of *ff* rules for Make.p.o control the compilation of the main source for Make, with some interesting side effects. The first *ff* rule saves the Make source before it is compiled, only if the source file has changed. The second *ff* rule does the actual compile. Note that this last rule has no explicit build commands, so it will be augmented by the built-in default rules, which will add a dependency relation (on the source file Make.p), and will supply the actual build commands for the compile.

8. The {ObjDir} prefix is necessary for the directory dependency rule to take effect. It allows the object and source to be in different directories.

9. The dependency rules for MacInterfaces, MemMgr, SymMgr, Utilities, and IOInterfaces describe dependencies between various utility units used by Make. Several dependencies on library interface files are given. Dependencies among the utility units themselves are described by indicating a dependency on the object files of the lower-level (predecessor) units. These dependencies could have been expressed as dependencies on the source files of the lower-level units (because it is the source files that are read in a Uses list). However, expressing these dependencies on the object files has the nice property of ensuring that the lower-level units have been successfully compiled before the higher-level units are built.

10. The Backup, Restore, and Listings targets are additional **roots** (top-level targets) in Make's makefile, and thus represent other things that can be built besides MakeX itself. Note that the Backup and Restore targets do not actually get built by their build rules; they are thus *artificial targets* and will always generate build commands if they are specified on the Make command line. Note also that they do not have any dependency relations.

11. The build rules for the Listings target demonstrates the use of the {NewerDeps} variable. The prerequisite of Listings is a list of the Make source files. The first build command prints the {NewerDeps} files. {NewerDeps} contains the names of the prerequisites that are newer than the target, that is, the source files that have changed since listings were last made. The last line of the build rules simply writes the current date into a file called Listings, which is the name of our target—this action results in a file that remembers when listings were last made. (Writing the datecvf into the file is unnecessary but convenient; the Echo itself is enough to change the file's last-modified date.)

◆ *Note:* There are several implicit builds that are generated as needed by the default rules. For example, the {MakeObjs} variable includes several assembly-language object files. Because {MakeObjs} appears as a prerequisite of the link step, these assemblies are generated, if necessary, before the link.

Chapter 10   More About Linking

THIS CHAPTER SUPPLEMENTS THE INFORMATION IN THE SECTION "LINKING" IN CHAPTER 8 and in the description of the Link command in Part II. This chapter will be more useful after you're familiar with Chapter 8 and the major MPW tools and when you are ready to optimize your programs or build procedures.

Use Link, the MPW linker, to combine a group of MPW object files (such as the output of the compilers) into a Macintosh-executable resource, such as an application, desk accessory, driver, or MPW tool. ■

Contents

Link functions

After a source file has been assembled or compiled into an object file, it contains

- Object code (relocatable machine language).

- Symbolic (named) references to all identifiers whose locations were not known at compile time. (These include references to routines from separate compilations and libraries, and references to global variables.)

The linker performs the following functions:

- Sorts code and data modules into segments, by segment name. (Within a segment, modules are placed in the order in which they occur in the input files.) The -sg and -sn options allow you to change segmentation at link time.

  ◆ *Note:* A **module** is a contiguous region of memory that contains code or static data. A module is the smallest unit of memory that is included or removed by the linker. A **segment** is a named collection of modules.

- Omits unused ("dead") code and data modules from the output file. (These modules can be listed with Link's -uf option, and deleted from libraries with the Lib command's -df option.)

- Provides (together with the Segment Loader) a jump table architecture that supports relocation of code and data at run time. (See the chapter "Segment Loader" of *Inside Macintosh* for more information about the jump table.)

- Constructs jump table entries only when needed, that is, only when a symbol is referenced across segments. This means that the jump table will be of minimum size.

- Edits instructions to use the most efficient addressing mode. A5-relative (jump table) addressing is used across segments, and PC-relative addressing is used within a segment.

  ▲ Warning   If you take the address of a procedure that is within the same segment, then, as stated, a PC-relative address is used to generate the effective address. (This is the case in MPW C by default, and in MPW Pascal when used with the -b option.) If the procedure address is stored as a variable (or passed to the tool box), and the segment is unloaded, then any routine calling that address will transfer control to the wrong place, with the result that the program will crash. See Macintosh Technical Note 42. ▲

△ **Important**   Note that the **-b** option in MPW Pascal means that you will use the A5 offset to the jump table, rather than the PC-relative address. The meaning of the **-b** option in MPW C is *opposite;* it forces PC-relative addressing and also places strings and constants in the same module. △

■ Provides (with the data initialization interpreter) support for relocation of data references at run time. (The **data initialization interpreter** is the module _DATAINIT in the libraries Runtime.o and CRuntime.o.)

■ Generates a cross-reference listing of link-time (object-level) names (**-x** option).

■ Generates a location map for debugging or performance analysis (**-map** option).

Link copies linked code segments into code resources in the resource fork of the output file. By default, these resources are given the same names as the corresponding segment names.

If linker errors or a user interrupt cause the output file to be invalid, the linker sets the file's modification date to "zero" (January 1, 1904, 12:00 A.M.). This action guarantees that the Make command will recognize that the file needs to be relinked, and that the MPW Shell will not run the file.

Segmentation

Segmenting a program makes it possible for temporarily unneeded parts of the program to be unloaded and purged from memory, thus freeing memory space. You specify the name of a segment by placing a directive in your program's source file. See the appropriate language reference manual for information. Each segment is linked into a separate code resource.

◆ *Note:* For a desk accessory or driver, the code must be in a single segment, and no jump table is constructed. Segmentation applies only to applications and MPW tools.

The linker sorts object code into load segments by name, allowing you to organize your source code for clarity and understanding. You can specify the same segment name more than once. Link collects code for a given segment name from all of Link's input files and places it into a single segment in the output file.

| ▲ Warning | Segment names are case sensitive. For example, "Seg1" and "SEG1" are not equivalent names. If you aren't sure about the cases used, you can use the linker's -p option to get a listing. ▲ |

By default, resources created by the linker are given resource names identical to the corresponding segment names. Link provides options for combining and renaming segments at link time (-sg and -sn). If you don't specify a segment name before the first routine in your file, the main segment name ("Main") is assumed there. Normally, you should give the main segment the name Main.

By default, segments are limited to 32,760 bytes. This limit ensures compatibility with all versions of the Macintosh ROM. Larger segments are allowed with Link's -ss option.

◆ *Note:* Object code is placed in a segment in the order that it's encountered in the input file. For segments larger than 32K, the order is important because PC-relative offsets are limited to 32K by MC68000 instructions.

For more information about segmentation, see the chapter "Segment Loader" of *Inside Macintosh.*

Segments with special treatments

When linking a main program, Link creates two segments that don't appear in the input object files:

■ The jump table ('CODE' resource, ID=0), which is unnamed.

■ The global data area (no resource), which is named %GlobalData and appears only in the link map file (described below). You can't change the name %GlobalData at link time.

There are also two segments that have special conventions:

■ The segment that contains the main program entry point ('CODE' resource, ID=1), usually named Main.

■ A segment named %A5Init, which contains the initial values for the global data area and code that moves these initial values to the global data area. Applications should unload this segment before allocating any memory in order to avoid memory fragmentation. You can unload the %A5Init segment by calling UnloadSeg with the address of entry point _DATAINIT as its parameter. In Pascal, for example,

```
UnloadSeg(&_DATAINIT);
```

In C, the same call looks like this:

```
UnloadSeg((Ptr)_DataInit);
```

In C and Pascal, this call should be the first statement in the application. In assembly language the call to UnloadSeg should follow the call to _DataInit.

Controlling the numbering of code resources

Normally, you don't need to worry about which segments are given which resource numbers. However, you may want to control the assignment of resource numbers to optimize program load time, to implement a specialized code manager, or to match the numbering produced by another linker.

Link creates and numbers code resources based on the order in which it encounters the segment names in the command-line parameters and the input object files. If you can't easily predict the order in which the names appear in the object files, you may want to force the ordering with command-line options that contain dummy segment-mapping directives. For example, the following sequence of linker options forces Main to come first, followed by Init, Body, and Term:

```
Link   -sn dummy1=Main   # must contain the main code module ∂
                         # or entry point ∂
       -sn %A5Init=Init ∂
       -sn dummy3=Body ∂
       -sn dummy4=Term ∂
       ...and so on
```

The "old" segment names may be either "dummy" names (which don't appear in the object files) or actual mappings, such as the mapping of the %A5Init code into the segment Init.

♦ *Note:* The segment containing the main code module will always be segment #1.

Resolving symbol definitions

This section describes how the linker resolves references to symbols. For a more detailed discussion of local and external symbols, see Appendix H.

Symbols in object files are either local or external. A **local** module or entry point can be referenced only from within the file where it is defined. An **external** module or entry point can be referenced from different object files. An **entry point** is a location (offset) within a module. (The module itself is treated as an entry point with offset zero.) A **reference** is a location within one module that will contain the address of another module or entry.

If the linker finds a symbol, it will first try to match the symbol to a local symbol in the same file. If the name cannot be located, the linker will then look for it externally. (An exception to this procedure is described in the "Record" section of Appendix H.)

Multiple external symbol definitions

If the object files contain more than one definition for an external symbol, the first definition is used, and all references are treated as references to the first definition. This lets you selectively override entry points in libraries so that you can substitute new versions of code. When subsequent definitions are encountered, a warning is generated.

♦ *Note:* If you override a module, then all succeeding entry points within the overridden module also disappear. Therefore be sure that any other referenced entry points in the overridden module are also defined in the new, overriding module.

Unresolved external symbols

Occasionally, you may find that an external symbol is unresolved because a reference was generated with case sensitivity set one way, whereas the definition was generated with different case rules. When this happens, you can avoid recompiling by using the Link option -ma (module alias). Whenever Link encounters an unresolved symbol, it checks the list of module aliases in an attempt to resolve it.

Building applications with more than 32K of global data

To permit your application to use more than 32K of global data, use the -m option of the MPW Pascal and MPW C compilers. The -m option generates code that causes global data references to be 32-bits. You should be aware that the code for 32-bit references is larger and slower than the code for 16-bit references.

Follow these steps:

1. Use the -m option when you compile Pascal or C files that reference "far" data. All Pascal units and the Pascal main (if any) in the program must be compiled using either -m or -n (see the note for Pascal users below).

2. Implement 32-bit references in assembly language when necessary (see the note for Assembler users below).

3. Use the linker's -srt option. This option instructs the linker to sort data modules into near and far groups, placing all 16-bit referenced global data as close to A5 as possible, and all only-32-bit-referenced data farther away. Thus, any data with a 16-bit reference is forced to within 32K of A5. You can also use the -ss *size* option to suppress the linker's warning about code or data segments larger than 32K.

The -srt option alters the usual ordering of global data (that is, it is no longer governed strictly by Link command line order).

32-bit references—MPW Pascal

If any Pascal unit in a program is compiled with -n or -m, then *all* Pascal units (including the Pascal main) must be compiled with either the -m or -n options. For units that don't need 32-bit references, specify -n.

Historically, Pascal global data was held in a single module (with the same name as the unit) and referenced by offsets into the module. The -n option generates a named module per data item (as in C). The -m option implies the -n option.

If you compile one unit (UNITA) with -n and another (UNITB) without -n, and if both units reference data declared in the other unit, this situation results:

Unit	has variables	compiled with	exports	references
UNITA	foo, bar	-n or -m	foo, bar	baz, bletch
UNITB	baz, bletch	*nothing*	_UNITB	_UNITA (+offset)

UNITA references data modules in UNITB using variable names (which are never defined), while UNITB references a module called _UNITA, which is never defined. The link will fail.

32-bit references—MPW Assembler

In assembly language you must explicitly code 32-bit references when you want to avoid fixing a data module to within 32K of A5. For the MC68000, you could write something like this:

```
IMPORT      LONGDATA:DATA
MOVE.L      indirect(PC),D0    ; [4/7/9 clocks]      offset -> scratch register
MOVE.x      (A5,D0.L),dest     ; [ea: 3/6/7 clocks]  access variable (PEA,etc.)
...

indirect:   DC.L      LONGDATA   ; 32-bit offset of data
```

In code that is intended to run only on a MC68020, you can do this:

```
MACHINE     MC68020
IMPORT      LONGDATA:DATA
MOVE.x      ((LONGDATA).L,A5),dest             ; move to destination (or PEA)
                                               ; [ea: 11/15/25 clocks]
```

◆ *Note:* The MC68020 code, while smaller, runs more slowly than the MC68000 code shown above if we ignore the possible impact of the temporary register required (11 versus 7 clocks best case, 15 versus 13 clocks cache case, and 25 versus 16 clocks worst case).

Linker location map

If you specify the Link option -map, Link writes a **location map** to standard output. The map contains information about segments and where modules are located in the segments. (See note later in this section about optional formats.)

For each code segment, the linker writes a segment map that looks like

```
Segment "Main" size=$0326 rsrcid=1 JTindex=$0000 #JTEnts=$0001
        COMPACTMEM          $0000   size=$0018 file="Interface.o"
        SAVE0RETA1          $000A
        SAVERETA1           $000C
        SAVE                $000E
        SAVE0               $0014
        NEWPTR              $0018   size=$000C
        CMain               $0024   size=$0036  JT=$0000(A5)
        _RTInit             $005A   size=$01F4 file="CRunTime.o"
        exit                $024E   size=$0020
        _RTExit             $026E   size=$0050
        c2pstr              $02BE   size=$0032
        p2cstr              $02F0   size=$001E
        main                $030E   size=$0018
```

■ The first line indicates the segment's name, size, and resource id. One or more module or entry point entries follow the segment description.

■ JTindex is the number of the segment's first jump table-entry.

■ JTEnts indicates the number of jump-table entries belonging to the segment.

■ A name of "%?Anon" indicates that the module or entry point is *anonymous* (was not given a name by the language processor).

■ The first number following the name is the module or entry point's segment offset. (If the segment is a 'CODE' segment, the segment offset doesn't include the 4-byte segment header required by the Segment Loader.)

■ If the entry is for a module, the module's size is indicated.

■ If the module or entry point has a jump-table entry, the A5-offset of that entry is indicated.

The name of the object file that the module came from is printed every time the object filename changes. That is, if subsequent modules come from the same object file, the object filename is not printed again (which reduces the size of the location map).

Map entries for the global data segment

When linking an application or tool with global data, Link writes a map of the global data segment that looks like:

```
Segment "%GlobalData" size=$0106
        #0001              -$0106(A5)   size=$000C
        _ArgC              -$0106(A5)
        _ArgV              -$0102(A5)
        _EnvP              -$00FE(A5)
        StandAlone         -$00FA(A5)   size=$0004   hasContents
        _IntEnv            -$00F6(A5)   size=$0034   hasContents
        _SAGlbls           -$00C2(A5)   size=$00BE   hasContents
        foo                -$0004(A5)   size=$0004   hasContents
```

■ The first line summarizes the global data segment, giving only its name and size.

■ Subsequent lines indicate the A5-offsets of variables.

■ If a line describes a module, the module's size is indicated; if there is no size present, the line describes an entry point within the module immediately above.

■ If the data module contains initialized data, the word 'hasContents' follows the size.

As for code segments, the name of the object file the data module came from is printed whenever the object file name changes.

Optional map formats for compatibility

The options -l, -la, and -lf produce a linker map in an obsolete format. This format has been retained only for compatibility with the MPW Performance Tools, which currently read the map files to determine module locations. Tools should *not* depend on the format of the location map, as it is likely to change in future releases of MPW. (If tools need information about module locations, they should read symbolic information files produced with the -Sym option. Documentation on the Sym file format is available separately from Developer Technical Support.)

Optimizing your links

Because of the complexity of the linker's functions, the Link step is often the longest single step during incremental rebuilding of your program. The following steps can substantially speed up Link's performance:

■ *Use a RAM cache.* Link must open and close many object files. Experience has shown that large links run up to four times faster when you use a RAM cache of 64K or more on machines with at least 1 megabyte or more of RAM. (Use the Control Panel desk accessory to check your RAM cache settings. If you change the setting, you must restart the system to have the new setting take effect.)

■ *Use the Lib utility to combine input files.* You can use the Lib command to reduce the number of input files. Using Lib can give a 10–15 percent improvement in linking speed. See "Library Construction" later in this chapter.

■ *Eliminate unneeded files.* You can eliminate unneeded input to the linker by heeding the warning "File not needed for link," and removing such files from the link. This means customizing your link lists, rather than relying on a generic makefile for linking.

■ *Eliminate unneeded references.* You can also eliminate unneeded input by using Lib to remove unreferenced modules. Experience has shown that producing a specialized library file can increase Link's speed by as much as 40 percent. See the next section, "Library Construction."

Library construction

The Lib tool enables library construction by allowing you to combine object code from different files and languages into a single object file. For example, you can combine assembly-language code with C or Pascal. The Lib tool was used for this purpose in constructing the libraries distributed with MPW.

The tool Lib and its options are described in Part II. This section explains some aspects of using Lib.

Lib reorganizes the input files, placing the combined library file in the data fork of the output library file. By default, the library output file is given type ' obj ' and creator 'mps '. Lib's output is logically equivalent to the concatenation of the input files, except for its optional renaming, resegmentation, and deletion operations, and the possibility of overriding an external name (as in Link). Lib *doesn't* combine modules into larger modules, nor does it resolve cross-module references. This limitation guarantees that the output of a link that uses the output of Lib is the same as that of a link using the "raw" files produced by the compilers and assembler.

◆ **Why lib can speed up your links**

Object files processed with Lib result in significantly faster links than the `raw` object files produced by the compilers and assembler. There are several reasons for the speed improvements:

■ Code and Data modules are separated into different sections, and Code modules are further sorted by segment name. These actions improve the performance of Link, which must sort input modules into output code resources.

■ All of the named symbols in the object file are gathered into a single Dictionary area at the start of the file. This makes the output file smaller and simplifies the processing needed by Link to resolve references.

   ■ When several object files are combined, multiple instances of a symbol definition are replaced by a single definition. Again, this makes the output file smaller and simplifies the processing by Link. ◆

Lib correctly handles file-relative scoping conventions, such as nested procedures in Pascal, static functions in C, or ENTRY names in assembly language; that is, it never confuses references to an external symbol with references to a local symbol of the same name, even if the two symbols are in two files combined with Lib.

Using Lib to build a specialized library

Each of the language libraries has files that you may or may not need to link with, depending on the functions your program calls. (See Appendix A.) Once you determine which files are needed for linking a particular program, you can greatly improve the performance of subsequent links by combining libraries into a single specialized library file. In building a specialized library, you can use Lib to

- change segmentation (with the -sg and -sn options)

- change the scope of a symbol from external to local (with the -dn option)

- delete unneeded modules (with the -dm option)

Lib's renaming, resegmentation, and deletion operations give you precise control over external names, the contents of library files, and the segmentation of object code. To use these features, you may need to review some of the material in Appendix H to understand how modules and entry points are represented in object files. The DumpObj command is also useful in exploring the contents and structure of the library files provided with MPW.

Removing unreferenced modules

You can eliminate unneeded input to Link by using Lib to remove unreferenced modules. You can determine the number of unreferenced modules from Link's progress information. (Use the -p option.) Link reports the total number of symbols read, as well as the number of active symbols (that is, the symbols in the output), and the number of visible symbols (that is, the symbols requiring jump-table entries)—for example,

```
155 active and 54 visible entries of 714 read.
```

The difference between the total read and the number of active symbols is the number of unreferenced (and unneeded) symbols. Most of these unreferenced symbols represent standard library functions that your particular program doesn't require.

Unreferenced modules can be removed in three steps:

1. Use Link's -uf option to produce a file containing the unreferenced names.

2. Use the -uf file produced by Link as the input to Lib, using the Lib option -df to produce a specialized library that contains only the modules that your program requires.

3. Use the output of Lib as the input to subsequent links.

Guidelines for choosing files for a specialized library

The choice of files to include in a specialized library file is largely dictated by stability issues. Files that are unlikely to change for many builds are the best candidates. Stable files include the library files provided by Apple for the ROM interfaces and for language support. Files that are under development are best left as single files.

Should you find it necessary to change one of the component files of a specialized library, you don't always need to rebuild the specialized library immediately. You can simply include the newer version of the object file in the link list by placing it before the specialized library file that contains the older version. You'll get some warning messages about duplicate symbols, but all references will be correctly moved to the first definition encountered by Link. Later, after the file is stable again, you can rebuild the library.

Chapter 11 Resource Compiler and Decompiler

THIS CHAPTER EXPLAINS THE USE OF THE RESOURCE COMPILER (REZ) AND RESOURCE decompiler (DeRez). The command line syntax for Rez and DeRez is explained in Part II. The general syntax for resource description files is summarized in Appendix D. You can build a resource in text form by using Rez, or graphically by using the application ResEdit. Complete background information on Macintosh resource files is given in the chapter "Resource Manager" of *Inside Macintosh.* ■

Contents

About the resource compiler and decompiler

The resource compiler, Rez, compiles a text file (or files) called a **resource description file** and produces a resource file as output. The resource decompiler, DeRez, decompiles an existing resource, producing a new resource description file that can be understood by Rez. Figure 11-1 illustrates the complementary relationship between Rez and DeRez.

■ **Figure 11-1**   Rez and DeRez
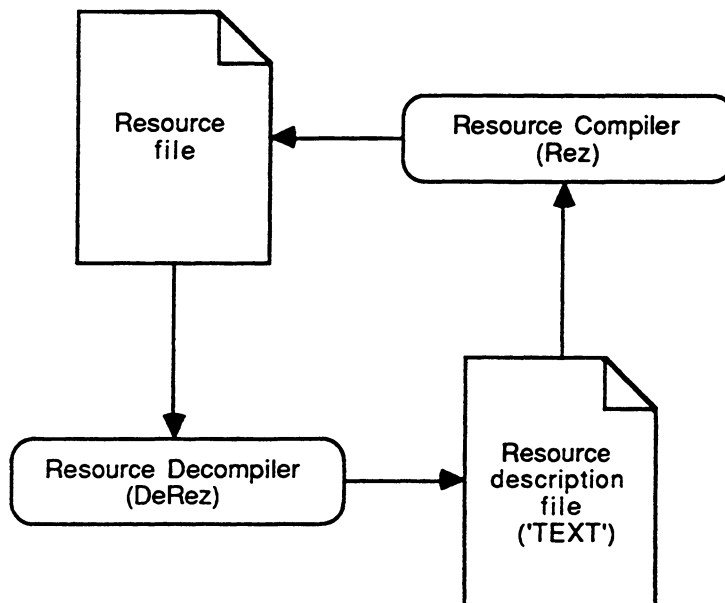
Rez can combine resources or resource descriptions from a number of files into a single resource file. Rez can also delete resources or change resource attributes. Rez supports preprocessor directives that allow you to substitute macros, include other files, and use if-then-else constructs. (These are described under the heading "Preprocessor Directives" later in this chapter.)

Resource decompiler

The DeRez tool creates a textual representation of a resource file based on resource type declarations identical to those used by Rez. (If you don't specify any type declarations, the output of DeRez takes the form of raw data statements.) The output of DeRez is a resource description file that may be used as input to Rez. This file can be edited in the MPW Shell, allowing you to add comments, translate resource data to a foreign language, or specify conditional resource compilation by using the if-then-else structures of the preprocessor. You can also compare resources by using the MPW Compare command to compare resource description files.

◆ *Note:* MPW includes a tool, ResEqual, which directly compares resource files. The Pascal source for ResEqual is located in the PExamples folder. Also see the MPW tool RezDet, described in Part II.

Standard type declaration files

Four text files, Types.r, SysTypes.r, MPWTypes.r, and Pict.r, contain resource declarations for standard resource types. These files are located in the {RIncludes} directory, which is automatically searched by Rez and DeRez (that is, you can specify a file in {RIncludes} by its simple filename).These files contain definitions for the following types:

Types.r   Type declarations for the most common Macintosh
             resource types ('ALRT', 'DITL', 'MENU', and so on)

SysTypes.r   Type declarations for 'DRVR', 'FOND', 'FONT', 'FWID', 'INTL',
             'NFMT', and many others

MPWTypes.r   Type declarations for the MPW driver type 'DRVW'

Pict.r   Type declaration for PICT resources for debugging PICTs

Cmdo.r   Type declaration for Commando resources

Using Rez and DeRez

Rez and DeRez are primarily used to create and modify resource files. Figure 11-2 illustrates the process of creating a resource file.

■ **Figure 11-2**   Creating a resource file
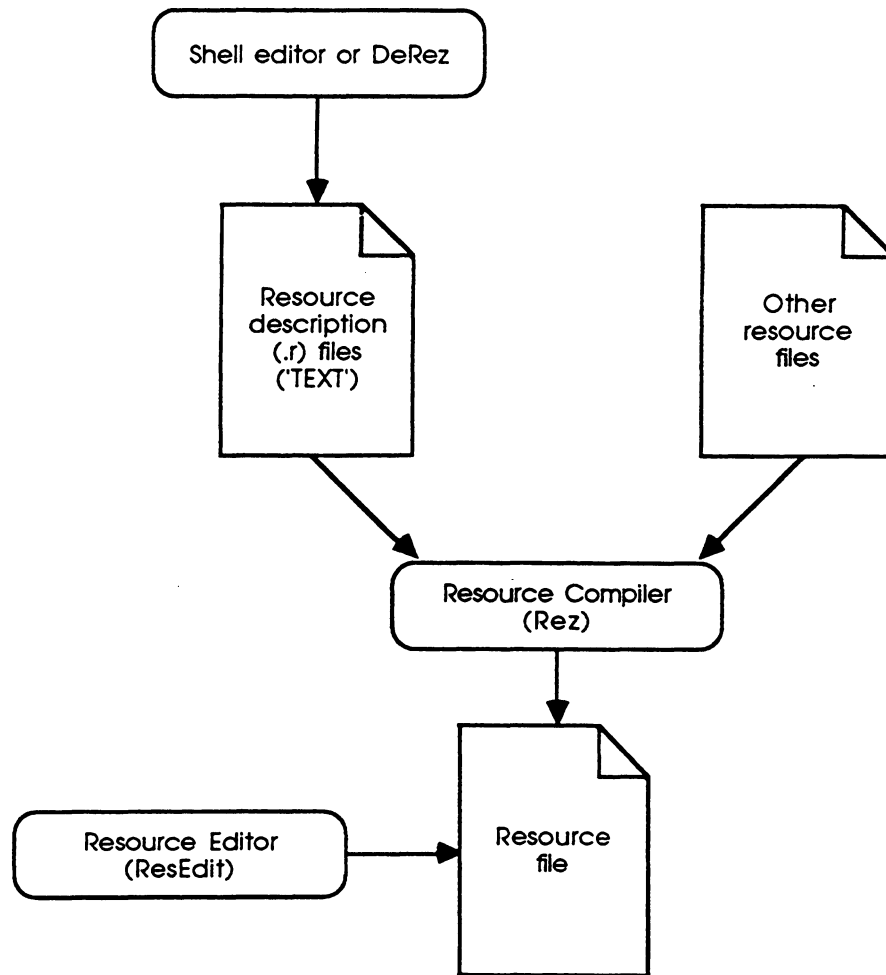
Rez can also form an integral part of the process of building a program. For instance, when putting together a desk accessory or driver, you would use Rez to combine the linker's output with other resources, creating an executable program file. (See Chapter 8 for details on building desk accessories and drivers.)

Structure of a resource description file

The resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler have no built-in resource types. You need to define your own types or include the appropriate ".r" files.

A resource description file may contain any number of these statements:

`include`	Include resources from another file.
`read`	Read data fork of a file and include it as a resource.
`data`	Specify raw data.
`type`	Type declaration—declare resource type descriptions for subsequent `resource` statements.
`resource`	Data specification—specify data for a resource type declared in a previous `type` statement.
`change`	Change the `type`, ID, name, or attributes of existing resources.
`delete`	Delete existing resources.

Each of these statements is described in the sections that follow.

A **type declaration** provides the pattern for any associated resource data specifications by indicating data types, alignment, size and placement of strings, and so on. You can intersperse type declarations and data in the resource description file as long as the declaration for a given resource precedes any resource statements that refer to it. An error is returned if data (that is, a resource statement) is given for a type that has not been previously defined. Whether a type was declared in a resource description file or in an `include` file, you can redeclare it by providing a new declaration later in a resource description file.

A resource description file can also include comments and preprocessor directives:

- **Comments** can be included any place white space is allowed in a resource description file, by putting it within the comment delimiters `/*` and `*/`. Note that comments do not nest. For example, this is *one* comment:

  ```
  /* Hello /* there */
  ```
  Rez also supports C++ style comments:

  ```
  type 'tost' {   // the rest of this line is ignored
  ```

- **Preprocessor directives** substitute macro definitions and `include` files, and provide if-then-else processing before other Rez processing takes place. The syntax of the preprocessor is very similar to that of the C-language preprocessor.

Sample resource description file

An easy way to learn about the resource description format is to decompile some existing resources. For example, the following command decompiles only the 'WIND' resources in the Sample application, according to the type declaration in {RIncludes}Types.r.

```
DeRez Sample -only WIND  Types.r  > DeRez.Out
```

Note that Rez automatically finds Types.r in {RIncludes}. After executing this command, DeRez.Out would contain the following decompiled resource:

```
resource 'WIND' (128, "Sample Window") {
        {64, 60, 314, 460},
        documentProc,
        visible,
        noGoAway,
        0x0,
        "Sample Window"
};
```

Note that this statement is identical to the resource description in the file Sample.r, which was originally used to build the resource. This resource data corresponds to the following type declaration, contained in Types.r:

```
type 'WIND' {
        rect;                                           /* boundsRect */
        integer documentProc, dBoxProc, plainDBox,     /* procID */
                altDBoxProc, noGrowDocProc,
                zoomProc=8, rDocProc=16;
        byte    invisible, visible;                     /* visible */
        fill byte;
        byte    noGoAway, goAway;                       /* goAway */
        fill byte;
        unsigned hex longint;                           /* refCon */
        pstring                                         Untitled = "Untitled";
        /* title */
};
```

Type and resource statements are explained in detail in the reference section that follows.

Resource description statements

This section describes the syntax and use of the seven types of resource description statements available for the resource compiler: include, read, data, type, delete, change, and resource.

Syntax notation

The syntax notation in this chapter follows the conventions given in the preface of this book. In addition, the following conventions are used:

■ Words that are part of the resource description language are shown in the Courier font (following the conventions used in documentation of the C language) to distinguish them from surrounding text. Rez is not sensitive to the case of these words.

■ Punctuation characters such as commas (,), semicolons (;), and quotation marks (' and ") are to be written as shown. If one of the syntax notation characters (for example, [or]) must be written as a literal, it is shown enclosed by "curly" single quotation marks ('...'); for example,

    bitstring '[' *length* ']'

   In this case, the brackets would be typed literally—they do *not* mean that the enclosed element is optional.

■ Spaces between syntax elements, constants, and punctuation are optional; they are shown for readability only.

Tokens in resource description statements may be separated by spaces, tabs, returns, or comments.

Special terms

The following terms represent a minimal subset of the nonterminal symbols used to describe the syntax of commands in the resource description language:

Term	Definition
resource-type	*long-expression*
resource-name	*string*
resource-ID	*word-expression*
ID-range	*ID* : *ID*

◆ *Note: Expression* is defined later in this chapter under "Expressions."

A full syntax definition can be found at the end of this chapter and in Appendix D.

Include—include resources from another file

The include statement lets you read resources from an existing file and include all or some of them.

Syntax

An `include` statement can take the following forms:

- `include` *file* [*resource-type* [' (' ' *resource-name* | *ID* [*:ID*]') ']] ;

  Read the resource of type *resource-type* with the specified resource name or resource ID range in *file*. If the resource name or ID is omitted, read all resources of the type *resource-type* in *file*. If *resource-type* is omitted, read all the resources in *file*.

- `include` *file* `not` *resource-type* ;

  Read all resources **not** of the type *resource-type* in *file*.

- `include` *file* *resource-type1* `as` *resource-type2* ;

  Read all resources of type *resource-type1* and include them as resources of *resource-type2*.

- `include` *file* *resource-type1* ' (' ' *resource-name* | *ID* [*:ID*]') '

  `as` *resource-type2* ' (' *ID* [, *resource-name*] [, *attributes...*]') ' ;

  Read the resource of type *resource-type1* with the specified name or ID range in *file*, and include it as a resource of *resource-type2* with the specified ID. You can optionally specify a resource name and resource attributes. (Resource attributes are defined below.)

Some examples follow:

```
include "otherfile"; /* include all resources from the file */
include "otherfile" 'CODE'; /* read only the CODE resources */
include "otherfile" 'CODE' (128); /* read only CODE resource 128 */
```

AS resource description syntax

The following string variables can be used in the `as` resource description to modify the resource information in include statements:

`$$Type`	Type of resource from include file
`$$ID`	ID of resource from include file
`$$Name`	Name of resource from include file
`$$Attributes`	Attributes of resource from include file

For example, to include all `'DRVR'` resources from one file and keep the same information but also set the `SYSHEAP` attribute:

```
INCLUDE  "file"  'DRVR'  (0:40)  AS
                 'DRVR'  ($$ID, $$Name, $$Attributes  |  64) ;
```

The `$$Type`, `$$ID`, `$$Name`, and `$$Attributes` variables are also set and legal within a normal resource statement. At any other time the values of these variables are undefined.

Resource attributes

You can specify **attributes** as a numeric expression (as described in the chapter "Resource Manager" of *Inside Macintosh*), or you can set them individually by specifying one of the keywords from any of the following pairs:

Default	Alternative	Meaning
appheap	sysheap	Specifies whether the resource is to be loaded into the application heap or the system heap.
nonpurgeable	purgeable	Purgeable resources can be automatically purged by the Memory Manager.
unlocked	locked	Locked resources cannot be moved by the Memory Manager.
unprotected	protected	Protected resources cannot be modified by the Resource Manager.
nonpreload	preload	Preloaded resources are placed in the heap as soon as the Resource Manager opens the resource file.
unchanged	changed	Tells the Resource Manager whether a resource has been changed. Rez does not allow you to set this bit, but DeRez will display it if it is set.

Bits 0 and 7 of the resource attributes are reserved for use by the Resource Manager and cannot be set by Rez, but are displayed by DeRez.

You can specify more than one attribute by separating the keywords with a comma (,).

Read—read data as a resource

The read statement lets you read a file's data fork as a resource.

Syntax

read *resource-type* ' (' *ID* [, *resource-name*] [, *attributes* ') ' *file* ;

Description

Reads the data fork from *file* and writes it as a resource with the type *resource-type* and the resource ID *ID*, with the optional resource name *resource-name* and optional resource attributes (as defined in the preceding section). For example,

```
read 'STR ' (-789,"Test String",SysHeap,PreLoad) "Test8";
```

Data—specify raw data

Use the data statement to specify raw data as a sequence of bits, without any formatting.

Syntax

```
data resource-type '(' ID[, resource-name] [, attributes...]')' '{'
      data-string
'}';
```

Description

Reads the data found in *data-string* and writes it as a resource with the type *resource-type* and the ID *ID*. You can optionally specify a resource name, resource attributes, or both.

For example,

```
data 'PICT' (128) {
      $"4F35FF8790000000"
      $"FF234F35FF790000"
};
```

◆ *Note:* When DeRez generates a resource description, it uses the data statement to represent any resource type that doesn't have a corresponding type declaration or cannot be disassembled for some other reason.

Type—declare resource type

A type declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one type declaration is given for a resource type, the last one read before the data definition is the one that's used. This lets you override declarations from include files or previous resource description files.

Syntax

```
type resource-type ['(' ID-range')'] '{'
      type-specification...
'}';
```

Description

Causes any subsequent resource statement for the type *resource-type* to use the declaration { *type-specification...* }. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one of the following:

```
bitstring [n]
byte
integer
longint
boolean
char
string
pstring
wstring
cstring
point
rect
fill          Zero fill
align         Zero fill to nibble, byte, word, or long word boundary
switch        Control construct (case statement)
array         Array data specification—zero or more instances of
              data types
```

These types can be used singly or together in a `type` statement. Each of these type specifiers is described in the sections that follow.

◆ *Note:* Several of these types require additional fields. The exact syntax is given in the sections that follow.

You can also declare a resource type that uses another resource's `type` declaration by using the following variant of the `type` statement:

`type` *resource-type1* [' (' *ID-range* ')'] `as` *resource-type2* ['(' *ID* ')'];

Data-type specifications

A Data-type statement declares a field of the given data type. It can also associate symbolic names or constant values with the data type. The `data-type` specification can take three forms, as shown in this example:

```
type  'XAMP' {       /* declare a resource of type 'XAMP' */
      byte;
      byte        off=0, on=1;
      byte = 2;
};
```

- The first `byte` statement declares a byte field; the actual data is supplied in a subsequent `resource` statement.

- The second `byte` statement is identical to the first, except that the two symbolic names "off" and "on" are associated with the values 0 and 1. These symbolic names could be used in the `resource` data.

- The third `byte` statement declares a byte field whose value is always 2. In this case, no corresponding statement would appear in the `resource` data.

Numeric expressions and strings can appear in `type` statements; they are defined later in this chapter under "Expressions."

Numeric types: The numeric types (`bitstring`,`byte`,`integer`,`longint`)are fully specified like this:

[*unsigned*] [*radix*] *numeric-type* [*=expr* | *symbol-definition...*];

- The Unsigned prefix signals DeRez that the number should be displayed without a sign—that the high-order bit can be used for data and the value of the integer cannot be negative. The Unsigned prefix is ignored by Rez but is needed by DeRez to correctly represent a decompiled number. Rez uses a sign if it is specified in the data. Precede a signed negative constant with a minus sign (–); $FFFFFF85 and –$7B are equivalent in value.

- *Radix* is one of the following string constants:

  hex   decimal   octal   binary   literal

  You can supply numeric data as decimal, octal, hexadecimal, or literal data.

- *Numeric-type* is one of the following:

`bitstring` [*length*]'	Declare a bitstring of *length* bits (maximum 32).
`byte`	Declare a byte (8-bit) field. This is the same as `bitstring[8]`.
`integer`	Integer (16-bit) field. This is the same as `bitstring[16]`.
`longint`	Long integer (32-bit) field. This is the same as `bitstring[32]`.

Rez uses integer arithmetic and stores numeric values as integer numbers. Rez translates booleans, bytes, integers, and longints to bitstring equivalents. All computations are done in 32 bits and truncated.

An error is generated if a value won't fit in the number of bits defined for the type. The valid ranges for values of `byte`, `integer`, and `longint` constants are as follows:

Type	Maximum	Minimum
byte	255	−128
integer	65,535	−32,768
longint	4,294,967,295	−2,147,483,648

Boolean type: A Boolean is a single bit with two possible states: 0 (or `false`) and 1 (or `true`). (`True` and `false` are global predefined identifiers.) Boolean values are declared as follows:

`boolean` [= *constant* | *symbolic-value...*] ;

The type `boolean` declares a 1-bit field; this is equivalent to

`unsigned bitstring[1]`

◆ *Note:* This type is not the same as a Boolean variable as defined by Pascal.

Character type: Characters are declared as follows:

`char` [= *string* | *symbolic-value...*] ;

Type `char` declares an 8-bit field (this is the same as writing `string[1]`).

Here is an example:
```
type 'SYMB'  {
      char  dollar = "$",percent = "%";
};
resource 'SYMB' (128)  {
      dollar
};
```

String type: String data types are specified like this:

string-type [' [' *length* '] '] [= *string* | *symbol-value...*];

String-type is one of the following:

`[hex] string`	Plain string (no length indicator or termination character is generated). The optional `hex` prefix tells DeRez to display it as a hex string. `String[n]` contains n characters and is n bytes long. The type `char` is shorthand for String `[1]`.
`pstring`	Pascal string (a leading byte containing the length information is generated). `Pstring[n]` contains n characters and is $n+1$ bytes long. `Pstring` has a built-in maximum length of 255 characters, the highest value the length byte can hold. If the string is too long to fit the field, a warning is given and the string is truncated.
`wstring`	Word string is a very large `pstring`. Its length is stored in the first two bytes. Therefore, a word string can contain up to 65,535 characters. `Wstring[n]` contains n characters and is $n+2$ bytes long.
`cstring`	C string (a trailing null byte is generated). `cstring[n]` contains $n-1$ characters and is n bytes long. A C string of length 1 can be assigned only the value `""`, because `cstring[1]` has room only for the terminating null.

Each string type may be followed by an optional *length* indicator in brackets (`[n]`). *Length* is an expression indicating the string length in bytes. *Length* is a positive number in the range $1 \leq length \leq 2147483647$ for `string` and `cstring`, and in the range $1 \leq length \leq 255$ for `pstring`, and in the range $1 \leq length \leq 65535$ for `wstring`.

◆ *Note:* You cannot assign the value of a literal to a string type.

If no length indicator is given, a `pstring`, `wstring`, or `cstring` stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all string types are nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

▲ **Warning**	A null byte within a `cstring` is a termination indicator and may confuse DeRez and C programs. However, the full string, including the explicit null and any text that follows it, will be stored by Rez as input. ▲

Resource description statements: Point and rectangle types: Because points and rectangles appear so frequently in resource files, they have their own simplified syntax:

`point [ = `*point-constant* | *symbolic-value*... `];`
`rect [ = `*rect-constant* | *symbolic-value*... `];`

where

point-constant = '{'*x-integer-expr, y-integer-expr* '}'

and

rect-constant = '{'*integer-expr*, *integer-expr*, *integer-expr*, *integer-expr*'}'

These type statements declare a point (two 16-bit signed integers) or a rectangle (four 16-bit signed integers). The integers in a rectangle definition specify the rectangle's upper-left and lower-right points, respectively.

Fill and align types

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The `fill` and `align` type specifiers are two ways of padding fields so that they begin on a boundary that corresponds to the field type. `Align` is automatic and `fill` is explicit. Both `fill` and `align` generate zero-filled fields.

Fill specification: The `fill` statement causes Rez to add the specified number of bits to the data stream. The fill is always 0. The form of the statement is

`fill` *fill-size* ['['*length*']'] ;

where *fill-size* is one of the following strings:

bit   nibble   byte   word   long

These declare a fill of 1, 4, 8, 16, or 32 bits (optionally multiplied by the *length* modifier). *Length* is an expression ≤ 2147483647.

The following `fill` statements are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

The full form of a `type` statement specifying a fill might be:

`type 'XRES' {`*data-type specifications*; `fill bit[2];};`

◆ *Note:* Rez supplies zeros as specified by `fill` and `align` statements. DeRez does not supply any values for `fill` or `align` statements; it just skips the specified number of bits, or until data is aligned as specified.

Align specification: Alignment causes Rez to add fill bits of zero value until the data is aligned at the specified boundary. An alignment statement takes the following form:

`align` *align-size* ;

where *align-size* is one of these strings:

nibble   byte   word   long

Alignment pads with zeros until data is aligned on a 4-, 8-, 16-, or 32-bit boundary. This alignment affects all data from the point where it is specified until the next `align` statement.

Array type

An array is declared as follows:

[wide]   array [*array-name* | '['*length*']'] '{' *array-list* '}';

The *array-list*, a list of type specifications, is repeated zero or more times. The `wide` option outputs the array data in a wide display format (in DeRez)—the elements that make up the array-list are separated by a comma and space instead of a comma, return, and tab. Either *array-name* or [*length*] may be specified. *Array-name* is an identifier.

If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$countof`(*array-name*) function; otherwise DeRez will treat the array as an open-ended array. For example,

```
type   'STR#' {              /* define a string list resource */
       integer = $$Countof(StringArray);
       array StringArray {
             pstring;
       };
};
```

The `$$countof` function returns the number of array elements (in this case, the number of strings) from the resource data.

If [*length*] is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators. In this example, however, it may be a good idea to use semicolons as element separators:

```
type 'xyzy' {
       array Increment {
             integer = $$ArrayIndex(Increment);
       };
};
resource 'xyzy' (0) {
       {       /* zero elements */
       }
};
resource 'xyzy' (1) {
       {       /* two elements */
             ,
       }
};
```

```
resource 'xyzy' (3) {
      }         /* two elements */
            ;;
      }
};
/* The only way to specify one element in an array that has all
      constant elements, is to use a semicolon terminator.
/*
resource 'xyzy' (4) {
      {         /* one element */
            ;
      }
};
```

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is:

switch '{' *case-statement...* '}';

where a *case-statement* has this form:

case *case-name* : [*case-body* ;]...

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

key *data-type* = *constant*

Which `case` applies is based on the key value. For example,

```
type 'DITL' {                          /* dialog item list declaration from Types.r */
      ...type specifications...
      switch {                         /* one of the following */
      case Button:
            boolean         enabled, disabled;
            key bitstring[7] = 4;   /* key value */
            pstring;
      case CheckBox:
            boolean         enabled, disabled;
            key bitstring[7] = 5;   /* key value */
            pstring;
                        ...and so on.
      };
};
```

Sample type statement

The following sample `type` statement is the standard declaration for a `'WIND'` resource, taken from the Types.r file:

```
type 'WIND' {
        rect;                                                 /* bounds    */
        integer     documentProc, dBoxProc, plainDBox,        /* procID    */
                    altDBoxProc, noGrowDocProc,
                    zoomProc=8, rDocProc=16;
        byte        invisible, visible;                       /* visible   */
        fill byte;
        byte        noGoAway, goAway;                         /* close box */
        fill byte;
        unsigned hex longint;                                 /* refCon    */
        pstring     Untitled = "Untitled";                    /* title     */
};
```

The `type` declaration consists of header information followed by a series of statements, each terminated by a semicolon (;). The header of the sample window declaration is

```
type 'WIND'
```

The header begins with the Type keyword followed by the name of the resource type being declared—in this case, a window. You may specify a standard Macintosh resource type, as shown in the chapter "Resource Manager" of *Inside Macintosh*, or you may declare a resource type specific to your application.

The left brace [{] introduces the body of the declaration. The declaration continues for as many lines as necessary until a matching right brace} is encountered. You can write more than one statement on a line, and a statement may be on more than one line (like the Integer statement above). Each statement represents a field in the resource data. Recall that comments may appear anywhere where white space may appear in the resource description file; comments begin with /* and end with /* as in C.

Symbol definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

name = value [, *name = value*]...

For numeric data, the "= *value* " part of the statement can be omitted. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out—if *value* is omitted, it's assumed to be one greater than the previous value. (The value is assumed to be zero if it's the first value in the list.) This is true for bitstrings (and their derivatives, `byte`, `integer`, and `longint`). For example,

```
integer        documentProc, dBoxProc, plainDBox,
               altDBoxProc, noGrowDocProc,
               zoomProc=8, rDocProc=16;
```

In this example, the symbolic names documentProc, dBoxProc, plainDBox, altDBoxProc, and noGrowDocProc are automatically assigned the numeric values 0, 1, 2, 3, and 4.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value; for example,

```
integer        documentProc=0, dBoxProc=1, plainDBox=2, altDBoxProc=3,
               rDocProc=16, Document=0, Dialog=1, DialogNoShadow=2,
               ModelessDialog=3, DeskAccessory=16;
```

Delete—delete a resource

Sometimes you may want to delete a resource without switching to ResEdit. Some resource operations, such as those needed by "internationalizing" system disks and applications need to translate menu and dialog text, and hence require deleting or changing resources.

Syntax

`delete` *resource-type* [`'` (`'`*resource-name*| ID [:ID] `'`)`'`];

Description

Delete the resource of type *resource-type* from the output file with the specified resource name or resource ID range. If the resource name or ID is omitted, all resources of type *resource-type* are deleted.

◆ *Note:* Of course, the `delete` function is valid only when the **-a** (append) option is specified in the command line. It makes no sense to delete resources while creating a new resource file from scratch.

You can delete resources that have their protected bit set only if you use the **-ov** option.

Here is an example of an executable Shell command that deletes the `'ckid'` resource from a file:

```
echo "delete 'ckid';" | rez -a -o SomeTextFile
```

Change—change a resource's vital information

You can change a resource's vital information by using this function. Vital information includes the resource type, ID, name, attributes, or any combination of these at once.

Syntax

change *resource-type1* [`'('` *resource-name*| ID[`:`ID]`'`)`'`]
to *resource-type2* `'('`ID[`,` *resource-name*]   [`,` *attributes...*]`'`)`'`;

Description

Change the resource of type *resource-type1* from the output file with the specified resource name or resource ID range to a resource of type *resource-type2* with the specified ID. You can optionally specify a resource name and resource attributes. If the resource name or attributes are not specified, the name and attributes are not changed.

For example, here is a Shell command that sets the protected bit on all code resources in the file TestDA:

```
echo "change 'CODE' to $$type ($$Id,$$Attributes | 8);" ∂
     | rez -a -o TestDA
```

◆ *Note:* The `change` function is only valid when the -a (append) option is specified in the command line. It makes no sense to change resources while creating a new resource file from scratch.

Resource—specify resource data

`Resource` statements specify actual resources, based on previous `type` declarations.

Syntax

```
resource resource-type '(' ID [, resource-name] [, attributes] ')' '{'
        [ data-statement [ , data-statement ]... ]
'}';
```

Description

Specifies the data for a resource of type *resource-type* and ID *ID*. The latest `type` declaration declared for *resource-type* is used to parse the data specification. *Data-statements* specify the actual data; *data-statements* appropriate to each resource type are defined in the next section.

The `resource` definition causes an actual resource to be generated. A `resource` statement can appear anywhere in the resource description file, or even in a separate file specified on the command line or as an `#include` file, as long as it comes after the relevant `type` declaration.

Data statements

The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

Base type	Instance types
`string`	`string, cstring, pstring, wstring, char`
`bitstring`	`boolean, byte, integer, longint, bitstring`
`rect`	`rect`
`point`	`point`

Switch data: Switch data statements are specified by using this format:

switch-name data-body

For example, the following could be specified for the `'DITL'` type given earlier:

```
...
CheckBox { enabled, "Check here" },
...
```

Array data: Array data statements have this format:

`'{' [ array-element [ , array-element ]... ] '}'`

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the `'STR#'` resource defined earlier:

```
resource 'STR#' (280) {
    {       "this",
            "is",
            "a",
            "test"
    }
};
```

Sample resource definition

This section describes a sample resource description file for a window. (See the chapter "Window Manager" of *Inside Macintosh* for information about resources in windows.)

Here, again, is the `type` declaration given above under "Sample Type Statement":

```
type 'WIND'{
    rect;                                               /* bounds   */
    integer     documentProc, dBoxProc, plainDBox,      /* procID    */
                altDBoxProc, noGrowDocProc,
                zoomProc=8, rDocProc=16;
    byte        invisible, visible;                     /* visible   */
    fill byte;
    byte        noGoAway, goAway;                       /* close box */
    fill byte;
    unsigned hex longint;                               /* refCon   */
    pstring     Untitled = "Untitled";                  /* title    */
};
```

Here is a typical example of the window data corresponding to this declaration:

```
resource 'WIND' (128,"My window",appheap,preload) {    /* Status report window
    */
    {40,80,120,300},                          /* Bounding rectangle    */
    documentProc,                             /* documentProc etc..    */
    Visible,                                  /* Visible or Invisible */
    goAway,                                   /* GoAway or NoGoAway    */
    0,                                        /* Reference value RefCon */
    "Status Report"                           /* Title                 */
    };
```

This data definition declares a resource of type `'WIND'`, using whatever type declaration was previously specified for `'WIND'`. The resource ID is 128; the resource name is "My window." Because the resource name is represented by the Resource Manager as a `pstring`, it should not contain more than 255 characters. The resource name may contain any character including the null character ($00). The resource will be placed in the application heap when loaded, and it will be loaded when the resource file is opened.

The first statement in the window type declaration declares a bounding rectangle for the window:

```
rect;
```

The rectangle is described by two points: the upper-left corner and the lower-right corner. The points of a rectangle are separated by commas like this:

{ *top*, *left*, *bottom*, *right*}

An example of data for these coordinates is

```
{40,80,120,300}
```

Symbolic names: Symbolic names may be associated with particular values of a numeric type. Notice that a symbolic name is given for the data in the second, third, and fourth fields of the window declaration. For example,

```
integer      documentProc=0, dBoxProc=1, plainDBox=2,
             altDBoxProc=3, noGrowDocProc=4,
             zoomProc=8, rDocProc=16;        /* windowType */
```

This statement specifies a signed 16-bit integer field with symbolic names associated with the values 0 to 4 and 16. The values 0 through 4 need not be indicated in this case; if no values are given, symbolic names are automatically given values starting at 0, as explained previously.

In the sample window declaration, we gave the values True (1) and False (0) to two different byte variables. For clarity, we used those symbolic names in the window's resource data; that is,

```
visible,
goAway,
```

instead of their equivalents

```
TRUE,
TRUE,
```

or

```
1,
1,
```

Labels

Labels support some of the more complicated resources such as 'NFNT' and color QuickDraw resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels.

Syntax

```
label     ::=   character {alphanum}* ':'
character ::=   '_' | A | B | C ...
number    ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum  ::=   character | number
```

Description

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear on a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See "Declaring Labels Within Arrays" later in this chapter for more information.

The value of a label is always the offset, *in bits,* between the beginning of the resource and the position where the label occurs when mapped to the resource data. In this example,

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};

resource 'cool' (8) {
  "Neato"
}
```

the integer following the cstring would contain:

```
( len("Neato") [5] + null byte [1] ) * 8 [bits per byte] = 48.
```

Built-in functions to access resource data

In some cases, it is desirable to access the actual resource data that a label points to. Several built-in functions allow access to that data:

■ `$$BitField(label, startingPosition, numberOfBits)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label.*

■ `$$Byte(label)`
Returns the byte found at *label.*

■ `$$Word(label)`
Returns the word found at *label.*

■ `$$Long(label)`
Returns the longword found at *label.*

For example, the resource type `'STR'` could be redefined without using a `pstring`.
Here is the definition of `'STR'` from Types.r:

```
type 'STR' {
      pstring;
}
```

Here is a redefinition of `'STR '` using labels:

```
type 'STR ' {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop: ;
};
```

Declaring labels within arrays

Labels declared within arrays may have many values. For every element in the array, there is
a corresponding value for each label defined within the array. Use array subscripts to
access the individual values of these labels. The subscript values range from 1 to n where n
is the number of elements in the array. Labels within arrays that are nested in other arrays
require multidimensional subscripts. Each level of nesting adds another subscript. The
rightmost subscript varies most quickly. Here is an example:

```
type 'test' {
      integer = $$CountOf(array1);
      array array1 {
            integer = $$CountOf(array2);
            array array2 {
foo:              integer;
            };
      };
};
resource 'test' (128) {
      {
            {1,2,3},
            {4,5}
      }
};
```

In the above example, the label `foo` takes on these values:

```
            foo[1,1] = 32      $$Word(foo[1,1]) = 1
            foo[1,2] = 48      $$Word(foo[1,2]) = 2
            foo[1,3] = 64      $$Word(foo[1,3]) = 3
            foo[2,1] = 96      $$Word(foo[2,1]) = 4
            foo[2,2] = 112     $$Word(foo[2,2]) = 5
```

A new built-in function may be helpful in using labels within arrays:

$$ArrayIndex(*arrayname*)

This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label limitations

Keep in mind the fact that Rez and DeRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

◆ *Note:* To decompile (or "deRez") a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can only have only one undefined label:

```
type 'test' {
        /* In the expression below, start is defined, next is undefined.*/
start:      integer = next - start;
        /* In the expression below, next is defined because it was used
           in a previous expression, but final is undefined.*/
middle:     integer = final - next;
next: integer;
final:
};
```

Actually, Rez can compile types that have expressions containing more than one undefined label, but DeRez cannot decompile those resources and simply generates data resource statements.

◆ *Note:* The label specified in $$BitField(), $$Byte(), $$Word(), and $$Long() must occur lexically before the expression; otherwise, an error is generated.

Using labels: two examples

The first example shows the modified 'ppat' declaration using the new Rez labels. Boldface text in the example indicates everything that is different between the 2.0 and 3.0 versions of 'ppat' because of the use of labels. Without using labels, the whole end section of the resource would have to be combined into a single hex string (everything following the PixelData label). Using labels, the complete 'ppat' definition can be expressed in Rez language.

```
type 'ppat' {
        /* PixPat record */
        integer     oldPattern,          /* Pattern type        */
                                         newPattern,
                                         ditherPattern;
        unsigned longint = PixMap / 8;   /* Offset to pixmap     */
        unsigned longint = PixelData / 8;                         /* Offset
to data             */
        fill long;                       /* Expanded pixel image */
        fill word;                       /* Pattern valid flag   */
        fill long;                       /* expanded pattern     */
        hex string [8];                  /* old-style pattern    */
                                         /* PixMap record        */

PixMap:
        fill long;                       /* Base address            */
        unsigned bitstring[1] = 1;       /* New pixMap flag         */
        unsigned bitstring[2] = 0;       /* Must be 0               */
        unsigned bitstring[13];          /* Offset to next row      */
        rect;                            /* Bitmap bounds           */
        integer;                         /* pixMap vers number      */
        integer     unpacked;            /* Packing format          */
        unsigned longint;                /* size of pixel data      */
        unsigned hex longint;            /* h. resolution (ppi) (fixed)*/
        unsigned hex longint;            /* v. resolution (ppi) (fixed)*/
        integer chunky, chunkyPlanar, planar;                    /* Pixel
storage format    */
        integer;                         /* # bits in pixel         */
        integer;                         /* # components in pixel   */
        integer;                         /* # bits per field        */
        unsigned longint;                /* Offset to next plane    */
        unsigned longint = ColorTable / 8;                       /* Offset
to color table      */
        fill long;                       /* Reserved                */

PixelData:
        hex string [(ColorTable - PixelData) / 8];


ColorTable:
        unsigned hex longint;            /*  ctSeed            */
        integer;                         /*  transIndex        */
        integer = $$Countof(ColorSpec) - 1;    /*  ctSize      */
        wide array ColorSpec {
                integer;                 /*  value             */
                unsigned integer;        /*  RGB: red          */
                unsigned integer;        /*  green             */
                unsigned integer;        /*  blue              */
        };
};
```

Here is another example of a new resource definition with the new features in bold. In this example, the $$BitField() function is used to access information stored in the resource, in order to calculate the size of the various data areas added at the end of the resource. Without labels, all data would have to be combined into one hex string.

```
type 'cicn' {
        /* IconPMap (pixMap) record */
        fill long;                              /* Base address              */
        unsigned bitstring[1] = 1;              /* New pixMap flag           */
        unsigned bitstring[2] = 0;              /* Must be 0                 */
pMapRowBytes: unsigned bitstring[13];           /* Offset to next row        */
Bounds:rect;                                    /* Bitmap bounds             */
        integer;                                /* pixMap vers number        */
        integer     unpacked;                   /* Packing format            */
        unsigned longint;                       /* Size of pixel data        */
        unsigned hex longint;                   /* h. resolution (ppi) (fixed)*/
        unsigned hex longint;                   /* v. resolution (ppi) (fixed)*/
        integer     chunky, chunkyPlanar, planar; /* Pixel storage format */
        integer;                                /* # bits in pixel           */
        integer;                                /* # components in pixel     */
        integer;                                /* # bits per field          */
        unsigned longint;                       /* Offset to next plane      */
        unsigned longint;                       /* Offset to color table     */
        fill long;                              /* Reserved                  */

                                                /* IconMask (bitMap) record  */
        fill long;                              /* Base address              */
maskRowBytes: integer;                          /* Row bytes                 */
        rect;                                   /* Bitmap bounds             */

                                                /* IconBMap (bitMap) record  */
        fill long;                              /* Base address              */
iconBMapRowBytes: integer;                      /* Row bytes                 */
        rect;                                   /* Bitmap bounds             */
        fill long;                              /* Handle placeholder        */
```

```
        /* Mask data */
        hex  string  [$$Word(maskRowBytes)  *  ($$BitField(Bounds,  32,
16)  /*bottom*/
                - $$BitField(Bounds,  0,  16)  /*top*/)];

        /* BitMap data */
        hex  string  [$$Word(iconBMapRowBytes)  *
                ($$BitField(Bounds,  32,  16)/*bottom*/
                - $$BitField(Bounds,  0,  16)  /* top */)];

        /* Color Table */
        unsigned  hex  longint;          /* ctSeed              */
        integer;                         /* transIndex          */
        integer  =  $$Countof(ColorSpec)  -  1;               /*
ctSize          */
        wide  array  ColorSpec  {
                integer;           /* value         */
                unsigned  integer;  /* RGB:  red      */
                unsigned  integer;  /* green         */
                unsigned  integer;  /* blue          */
        };

        /* PixelMap data */
        hex  string  [$$BitField(pMapRowBytes,0,13)  *
                ($$BitField(Bounds,32,16)     /*  bottom        */
                - $$BitField(Bounds,  0,  16)  /*top*/)];
};
```

Preprocessor directives

Preprocessor directives substitute macro definitions and include files and provide
if-then-else processing before other Rez processing takes place.

The syntax of the preprocessor is very similar to that of the C-language preprocessor.
Preprocessor directives must observe these rules and restrictions:

- Each preprocessor statement must be expressed on a single line, beginning on a new
  line and terminated by a return character.

- The pound sign (#) must be the first character on the line of the preprocessor
  statement (except for spaces and tabs).

- *Identifiers* (used in macro names) may be letters (A-Z, a-z), digits (0-9), or the
  underscore character (_).

- Identifiers may be any length.
- Identifiers may not start with a digit.
- Identifiers are not case sensitive.

Variable definitions

The `#define` and `#undef` directives let you assign values to identifiers:

```
#define macro data
#undef macro
```

The `#define` directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the Rez escape character. For example,

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

(Quotation marks within strings must also be escaped.)

`#undef` removes the previously defined identifier *macro*. Macro definitions can also be removed with the **-undef** option on the Rez command line.

The following predefined macros are provided:

Variable	Value
true	1
false	0
rez	1 or 0 (1 if Rez is running, 0 if DeRez is running)
derez	1 or 0 (0 if Rez is running, 1 if DeRez is running)

Include directives

The `#include` directive reads a text file:

```
#include file
```

Include the text file *file*. The maximum nesting is to ten levels. For example,

```
#include $$Shell("MPW") "MyProject:MyTypes.r"
```

Note that the `#include` preprocessor directive (which includes a file) is different from the previously described `include` statement, which copies resources from another file.

If-Then-Else processing

These directives provide conditional processing:

```
#if expression
[ #elif expression ]
[ #else ]
#endif
```

◆ *Note: Expression* is defined later in this chapter. When used with the `#if` and `#elif` directives, *expression* may also include this expression:
   `defined` *identifier* `or defined'('` *identifier* `')'`

The following may also be used in place of `#if`:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource   'STR '   (199)   {
#ifdef English
      "Hello"
#elif defined (French)
      "Bonjour"
#elif defined (Thai)
      "Sawati"
#elif defined (Japanese)
      "Konnichiwa"
#endif
};
```

Print directive

The `#printf` directive is provided to aid in debugging resource description files:

```
#printf(formatString, arguments...)
```

The format of the `#printf` statement is exactly the same as the `printf` statement in the C language, with one exception: There can be no more than 20 arguments. This is the same restriction that applies to the `$$format` function. The `#printf` directive writes its output to diagnostic output. Note that the `#printf` directive *does not* end with a semicolon.

For example:

```
#define          Tuesday          3
#ifdef Monday
#printf("The day is Monday, day #%d\n", Monday)
#elif defined(Tuesday)
#printf("The day is Tuesday, day #%d\n", Tuesday)
#elif defined(Wednesday)
#printf("The day is Wednesday, day #%d\n", Wednesday)
#elif defined(Thursday)
#printf("The day is Thursday, day #%d\n", Thursday)
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The above file generates this text:

```
The day is Tuesday, day #3
```

Resource description syntax

This section describes the details of the resource description syntax. For a complete summary definition, see Appendix D.

Numbers and literals

All arithmetic is performed as 32-bit signed arithmetic. The basic constants are shown in Table 11-1

■ **Table 11-1**   Numeric constants

Numeric type	Form	Meaning
Decimal	*nnn...*	Signed decimal constant between 4,294,967,295 and –2,147,483,648.
Hex	0X*hhh...*	Signed hexadecimal constant between 0X7FFFFFFF and 0X80000000.
	$*hhh...*	Alternate form for hexadecimal constants.
Octal	0*ooo...*	Signed octal constant between 017777777777 and 020000000000.
Binary	0B*bbb...*	Signed binary constant between 0B1111111111111111111111111111111 and 0B10000000000000000000000000000000.
Literal	'*aaaa*'	A literal may contain one to four characters. Characters are printable ASCII characters or escape characters.If there are fewer than four characters in the literal, then the characters to the left (high bits) are assumed to be $00. Characters that are not in the printable character set, and are not the characters \ ' and \ \ (which have special meanings), can be escaped according to the character escape rules. (See "Strings" later in this section.)

Literals and numbers are treated in the same way by the resource compiler. A **literal** is a value within single quotation marks; for instance, 'A' is a number with the value 65; on the other hand, "A" is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that "A" is not a valid number and 'A' is not a valid string.) The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

Literals are padded with nulls on the left side so that the literal 'ABC' is stored as shown in Figure 11-3.

■ **Figure 11-3**   Padding of literals

"ABC" = | $00 | A | B | C |
| --- | --- | --- | --- |

Expressions

An expression may consist of simply a number or literal. Expressions may also include numeric variables, labels, and system functions.

Table 11-2 lists the operators in order of precedence with highest precedence first—groupings indicate equal precedence. Evaluation is always left to right when the priority is the same. Variables are defined following the table.

■ **Table 11-2**   Resource description expression operators

Operator	Meaning
1. (*expr*)	Parentheses can be used in the normal manner to force precedence in expression calculation
2. −*expr*	Arithmetic (two's complement) negation of *expr*
~*expr*	Bitwise (one's complement) negation of *expr*
! *expr*	Logical negation of *expr*
3. *expr1* * *expr2*	Multiplication
expr1 / *expr2*	Division
expr1 % *expr2*	Remainder from dividing *expr1* by *expr2*
4. *expr1* + *expr2*	Addition
expr1 − *expr2*	Subtraction
5. *expr1* << *expr2*	Shift left—shift *expr1* left by *expr2* bits
expr1 >> *expr2*	Shift right—shift *expr1* right by *expr2* bits
6. *expr1* > *expr2*	Greater than
expr1 >= *expr2*	Greater than or equal to
expr1 < *expr2*	Less than
expr1 <= *expr2*	Less than or equal to

(Continued)

■ **Table 11-2** (Continued)   Resource description expression operators

Operator	Meaning
7. *expr1* == *expr2*	Equal
expr1 != *expr2*	Not equal
8. *expr1* & *expr2*	Bitwise AND
9. *expr1* ^ *expr2*	Bitwise XOR
10. *expr1*	*expr2*	Bitwise OR
11. *expr1* && *expr2*	Logical AND
12. *expr1*		*expr2*	Logical OR

The logical operators !, >, >=, <, <=, ==, !=, &&, and | | evaluate to 1 (true) or 0 (false).

Variables and functions

Some resource compiler variables contain commonly used values. All Rez variables start with `$$` followed by an alphanumeric identifier.

The following variables and functions have string values (typical values are given in parentheses):

`$$Date`	Current date. Useful for putting timestamps into the resource file. The format is generated through the ROM call `IUDateString`. ("Thursday, May 20, 1987")
`$$Format ("formatString", arguments)`	Works just like the `#printf` directive except that `$$format` returns a string rather than printing to standard output. (See the section "Print Directive" earlier in this chapter.)
`$$Name`	Name of resource from the current resource. The current resource is the resource being generated in a `resource` statement, being included from an `include` statement, being deleted from a `delete` statement, or changed in a `change` statement.

For example, to include all `'DRVR'` resources from one file and keep the same information, but also set the SYSHEAP attribute:

```
INCLUDE  "file"  'DRVR'  (0:40)  AS
'DRVR  ($$ID, $$Name, $$Attributes | 64) ;
```

The `$$Type`, `$$ID`, `$$Name`, and `$$Attributes` variables are undefined outside of a `change`, `delete`, `include`, or `resource` statement.

`$$Resource` (*"filename"*, `'type'`, *ID* | *"resourceName"*)

Reads the resource `'type'` with the ID *ID* or the name *"resourceName"* from the resource file *"filename"*, and returns a string.

`$$Shell` (*"stringExpr"*)

Current value of the exported Shell variable {*stringExpr*}. Note that the braces must be omitted, and the double quotation marks must be present.

`$$Time`

Current time. Useful for time-stamping the resource file. The format is generated through the ROM call `IUTimeString`. ("7:50:54 AM")

`$$Version`

Version number of Rez. ("V3.0")

These variables and functions have numeric values:

`$$Attributes`

Attributes of resource from the current resource. See the `$$Name` string variable.

`$$BitField` (*label*, *startingPosition*, *numberOfBits*)

Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte` (*label*)

Returns the byte found at *label*.

`$$Day`

Current day. Range 1–31.

`$$Hour`

Current hour. Range 0–23.

`$$ID`

ID of resource from the current resource. See the $$Name string variable.

`$$Long` (*label*)

Returns the longword found at *label*.

`$$Minute`

Current minute. Range 0–59.

`$$Month`

Current month. Range 1–12.

`$$PackedSize` (*Start*, *RowBytes*, *RowCount*)

Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the ToolBox routine UnpackBits *RowCount* times. `$$PackedSize`( ) returns the unpacked size of the data found at *start*. Use this function only for decompiling resource files. An example of this function is found in Pict.r.

`$$ResourceSize`	Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource. (See the `'KCHR'` resource in SysTypes.r for an example.)
`$$Second`	Current second. Range 0–59.
`$$Type`	Type of resource from the current resource. See the `$$Name` string variable.
`$$Weekday`	Current day of the week. Range 1–7 (that is, Sunday–Saturday).
`$$Word(label)`	Returns the word found at *label*.
`$$Year`	Current year.

Strings

There are two basic types of strings:

Text string	`"a..."`	The string can contain any printable character except ' " ' and '\'. These and other characters can be created through escape sequences. (See Table 8-2.) The string `" "` is a valid string of length 0.
Hex string	`$"hh..."`	Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string `$" "` is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

Figure 11-4 shows a Pascal string declared as

`pstring [10];`

whose data definition is

`"Hello"`

- **Figure 11-4**   Internal representation of a Pascal string

$05	H	e	l	l	o	$00	$00	$00	$00	$00

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma) or brace signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks (" ") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash like this

(\ ").

Escape characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence

\n

Valid escape sequences are shown in Table 11-3.

■ **Table 11-3**   Resource compiler escape sequences

Escape sequence	Name	Hex value	Printable equivalent
\t	Tab	$09	None
\b	Backspace	$08	None
\r	Return	$0A	None
\n	Newline	$0D	None
\f	Form feed	$0C	None
\v	Vertical tab	$0B	None
\?	Rubout	$7F	None
\\	Backslash	$5C	\
\'	Single quotation mark	$3A	'
\"	Double quotation mark	$22	"

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

Base	Number form	Digits	Example
2	\0B*bbbbbbbb*	8	\0B01000001
8	*ooo*	3	\101
10	\0D*ddd*	3	\0D065
16	\0X*hh*	2	\0X41
16	\$*hh*	2	\$41

Here are some examples:

```
\077              /* 3 octal digits            */
\0xFF             /* '0x' plus 2 hex digits     */
\$F1\$F2\$F3      /* '$' plus 2 hex digits      */
\0d099            /* '0d' plus 3 decimal digits */
```

◆ *Note to C programmers:* An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write AB\007CD, not AB\7CD.

You can use the DeRez command line option -e to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between $20 and $D8 are printed as Macintosh characters. With this option, however, all characters (except null, newline, tab, backspace, form-feed, vertical tab, and rubout) will be printed as characters, not as escape sequences. See DeRez in Part II for details.

Chapter 12 **Writing an MPW Tool**

THIS CHAPTER PROVIDES INFORMATION SPECIFIC TO WRITING AN INTEGRATED MPW
TOOL. You'll find the utility routines used by tools that run within the MPW Shell
environment and how to access them. ■

Contents

Overview

This chapter provides information specific to writing an integrated MPW tool. You'll also need to refer to the following:

- Chapter 8, "The Build Process," for information about the mechanics of linking.
- Chapter 13, "Creating a Commando Interface for Tools."
- Tools Libraries in Appendix F. These contain the MPW Assembler, MPW C, and MPW Pascal routines for creating the rotating beach ball cursor and the Error Message File manager.
- The Graf3D Library in Appendix G. Your programs can use these routines to draw three-dimensional objects.

In this chapter you'll find the utility routines used by tools that run within the MPW Shell environment and how to access them. Examples of each of these routines are provided for MPW C, MPW Pascal, and MPW Assembly language. The MPW libraries contain four groups of these routines:

- Shell environment routines: procedures, functions, and data structures required to access MPW command-line parameters, Shell variables, and the standard input, output, and diagnostic files
- Shell signal-handling routines: procedures and functions that give you access to MPW software interrupts
- MPW cursor-control routines: procedures that let you control the form and action of the cursor. These are in Appendix F.
- Error message file management routines: procedures that let you access messages in the Macintosh system error message file. These are in Appendix F.

These routines provide tools running within the MPW Shell environment with many facilities, including:

- parameter passing
- access to Shell variables
- a set of preopened files for text-oriented input and output
- I/O to windows and selections
- a means for returning status results
- signal handling (for user aborts, and so on)
- exit processing

After introducing the conventions that MPW tools should follow and the specifics of linking tools, each of the sections that follow explain the use of these facilities for each of the MPW programming languages MPW C, MPW Pascal, and MPW Assembler.

Each section describes the environment in which the tool runs. It then lists the elements needed to access the libraries available to a tool. Finally, it lists the individual functions available to tools. Sections detailing each function are titled according to the standard C naming conventions and are written in this format:

name—short description of function

```
function prototytpe
```

Function description: What the routine does, how to use it, arguments needed.

MPW C

```
function prototype in MPW C
```

MPW Pascal

```
function prototype in MPW Pascal
```

Any applicable information for MPW Pascal.

MPW Assembler

```
function prototype in MPW Assembler
```

Any applicable information for MPW Assembler, including the language in which the function is written.

Conventions

MPW tools adhere to a certain style that allows them to work well together in an integrated fashion:

- Tools take their inputs as command-line parameters, rather than prompting for input. This input style allows their execution to be automated and allows them to take advantage of the Shell's command-line processing features such as variable substitution and filename generation.
- Deviations from a tool's standard behavior are specified with command options. Options may be specified anywhere on the command line and their order is not significant.

- Tools operate on a list of filename parameters, not just one, allowing the Shell's filename generation feature to be exploited.

- When no file parameters are given, tools take their input from standard input and write their output to standard output. The use of standard I/O allows the piping of the output of one program into the input of another. For example,

```
Files | Count -l
```

This command sends the output of the Files command into the input of the Count command, yielding the number of files and directories in the current directory.

- Tools spin the cursor to allow switching to different applications during tool execution (under MultiFinder). The cursor is spun at regular intervals for cooperative multitasking.

- Most tools operate silently as they process their input. Visual feedback is provided by the spinning cursor. If more feedback is desired, a -p (progress) option is usually provided to send status and summary information to the diagnostic output.

- Error messages are in the form of Shell comments or are "executable" so that the error can be easily located. For example, the language translators report errors in the form

```
File "Test.c" ; line 25   ### expected: ';' got: name
```

This message may be directly executed, to open the file and select the offending line. (See "Executable Error Messages" in Chapter 5.)

See the "Command Prototype" section at the beginning of Part II for more information on MPW command language conventions.

Status Codes

Every tool is expected to return a status code to the Shell when it terminates. The Shell inspects this result—if the status code is nonzero and if the Shell variable {Exit} is nonzero (the default), the Shell terminates the execution of the current command file. The Shell also converts the result to string form and creates a Shell variable called {Status} with that value. The variable can then be tested with the Shell command language and action can be taken based on its value.

The following conventions are used for status codes:

0	Success
1	Command syntax error
2	Some error in processing
3	System error or insufficent resources
-9	User abort

◆ *Note:* Only the bottom 24 bits of a tool's status code are returned to the Shell. All negative numbers, except for –9, are reserved for use by the Shell. See "Negative Status" in Chapter 5 for the meanings of negative status codes.

You may want to return error codes other than these. In that case, you should carefully document the numbers and their meanings.

MPW C	Result codes are passed as the return value from your main function or as the parameter to the C Library `exit` function.
MPW Pascal	Pascal programmers must call the IntEnv procedure IEexit to return the status result.
MPW Assembly	The Integrated Environment routine `_RTExit` is available to assembly-language programmers. `_RTExit` takes the status code as a parameter.

◆ *Note:* The returned status code will be undefined if you do not explicitly return a value by using the method recommended for your language.

Restrictions

Tools are similar to desk accessories in that they co-exist with another program (that is, the MPW Shell). The following sections touch on some of the considerations in enabling tools to co-exist with the Shell.

Initialization

Because tools run with the Shell, most Macintosh Toolbox initialization calls are not necessary and should not be called. In particular, you should *not* make the following calls:

```
InitFonts
InitWindows
InitMenus
TEInit
InitDialogs
MaxApplZone
SetApplLimit
SetGrowZone
InitResources
RsrcZoneInit
ExitToShell
```

(Note that this is not an inclusive list.)

If your tool uses QuickDraw or any routine that uses QuickDraw, be sure to call the `InitGraf` routine. This routine is necessary when using QuickDraw, because QuickDraw uses register A5-relative global variables, and tools have their own private A5 global area. Even a simple call to the QuickDraw function `Random` will not work properly unless `InitGraf` is called.

If your program opens any windows, make sure that it closes or disposes of those windows before it terminates.

◆ *Note:* If your tool calls `InitGraf` and writes to `stdout` or `stderr` (including error messages), then you should call `SetFScaleDisable` with a parameter value of `true` after your call to `InitGraf`. Otherwise your text output might be improperly scaled.

Memory Management

The Shell and tools execute out of the same heap and share the same stack. When a tool is started, the Shell allocates an area in the heap for the tool's globals and jump table, adjusts the global register A5 to point there, and then "calls" the tool. Any dynamic stack space required is allocated on the same stack, and any heap objects created go into the same heap.

■ **Figure 12-1**   Memory map


```
┌─────────────────────────┐
│         Screen          │──────────── A5—Globals (Shell)
├─────────────────────────┤
│      Shell globals      │
├─────────────────────────┤──────── A6—Stack frame pointer
│  │                      │
│  │      Stack          │
│  ▼                      │──────────── A7—Top of stack
├─────────────────────────┤
│░░░░░░░░░░░░░░░░░░░░░░░░░░│
│░░░░░░░░░░░░░░░░░░░░░░░░░░│
├─────────────────────────┤
│                         │
│          Heap           │
│   ┌─────────────────┐   │
│   │   Tool globals  │   │──────── A5—Globals (Tool)
│   └─────────────────┘   │
│                         │
│                         │
├─────────────────────────┤
│       System stuff      │
└─────────────────────────┘
```

Low memory
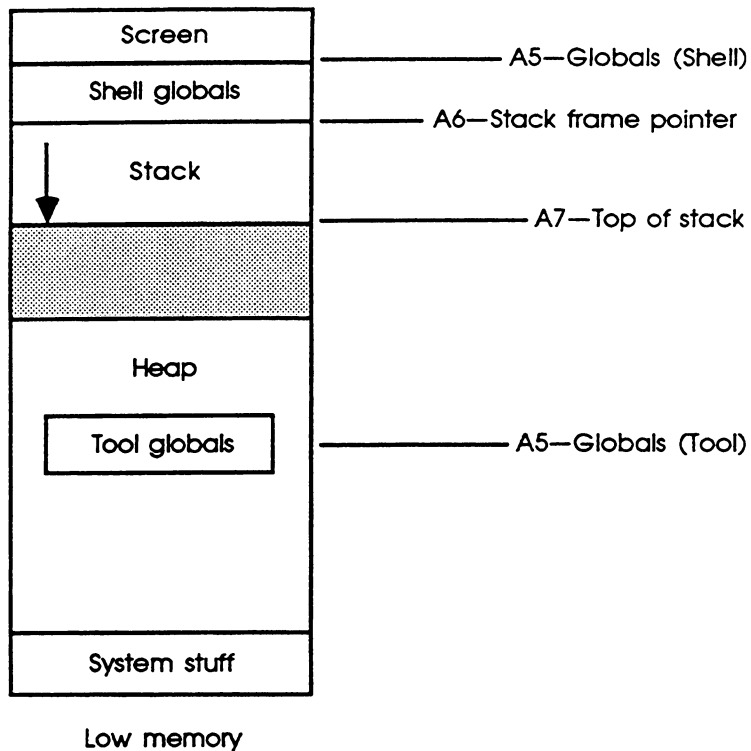
When a tool terminates, the Shell restores the registers to their previous values and deallocates the tool's global area and any other pointers and handles in the heap that may have not been allocated. The tool's resources, however, are not deallocated immediately. They are unlocked and made purgeable so that the space can be used if needed. This practice allows for a quick restart of the tool if it is still in memory, but with no memory wastage should the space be needed for other purposes.

▲ **Warning**   Although the Shell releases memory that has been allocated by the tool, sometimes the Shell has insufficient information to determine the owner of a master pointer. When a master pointer is NIL, it cannot be released by the Shell and cannot be reused.

NIL master pointers are produced as a result of calls to EmptyHandle, and by a number of Resource Manager actions. For example, a GetResource with ResLoad set to FALSE will create a NIL master pointer. If this is followed by a DetachResource or RmveResource, the handle remains as a NIL pointer. It is always good programming practice to clean up handles after they have become obsolete. Use DisposHandle to get rid of such obsolete handles. ▲

Heap

Because the Shell and tools share the same heap, some cooperation is necessary to ensure efficient use of the heap. Before a tool is started, the Shell makes many of its heap objects unlocked and purgeable. The Shell's memory-resident code is kept as low in the heap as possible. The tool's code should be moved as high in the heap as possible. This is done automatically, if the locked bit is not set on the tool's code resources (the default from the linker). When allocating heap space, tools should attempt to allocate no more space than is needed so that objects aren't needlessly purged from the heap.

When there is insufficient memory space to run a tool, you can make more space available in several ways.

To obtain more memory while running MPW:

- Close all MPW windows. (Certain memory-resident data structures are required for each window.)

- Pipe tool output to a file, rather than to a window.

- Your tool may be able to borrow memory from the MultiFinder heap when running MultiFinder.

To obtain more memory by relaunching MPW:

- If you are running MultiFinder, change the partition size for MPW in MPW's Get Info window.

- Change the HEXA Shell resource as described in the next section.

To obtain more memory by rebooting the Macintosh system:

- Turn off or reduce the size of the cache. (If you are running MultiFinder, you'll need to reboot to change the cache size.)

- Remove any debuggers from the system folder. You can free up about 90K by running without the MacsBug debugger (that is, hold down the mouse button while booting).

Here is the main difference between running under MultiFinder and Finder: Under MultiFinder, the amount of memory allocated to MPW is determined by the partition size (which you can change in the Info window). Under the Finder, available memory is affected by how much available system memory exists.

Stack

When the Shell starts up, it immediately grows the heap to its maximum size based on the maximum stack size. The default maximum dynamic stack size is 10K bytes when less than 480K is available for the application heap; the default maximum dynamic stack size is 20K when more than 480K is available. Because some tools may require more stack space or more heap space, 'HEXA' resource number 128 is available.

◆ *Note:* Because the stack is shared between the Shell and the tool, executing tools from within nested scripts results in less stack space for the tool. The Shell uses about 200 bytes of stack per nesting level.

▲ **Warning**   The MPW Shell segments might not be able to load into memory if:
1. Your tool calls `MaxMem` and
2. It allocates all available memory and
3. You then call any Shell services (such as writing to an open window). ▲

Building an MPW tool

In addition to traditional Macintosh applications, the Shell provides an environment for a type of program called an MPW tool. When a tool is run from the Shell, it does not replace the Shell or erase the screen, but instead runs within the Shell environment and has access to the facilities provided by the Shell. The compilers, Link, Make, and so on, are all tools in the MPW system.

From a programming viewpoint, tools resemble applications in many aspects of their behavior. Like applications, tools may have global variables and tools are linked just as applications are linked. The major difference between tools and applications is that tools do not have to initialize their environment (except for Quickdraw, if used) and tools have access to any of the Shell's open windows.

For a description of additional facilities available to an MPW tool, such as the Cursor Control and Error Message File Manager routines, see Appendix F. The Graf3d library is described in Appendix G.

Linking a tool

Linking an MPW tool is the same as linking an application (described in Chapter 8), except that the file type must be set to MPST and the creator to 'MPS ' (MPS*space*):

```
Link -t MPST  -c "MPS " ...
```

Sample tools are provided in the Examples folders for each of the MPW languages—refer to the sample makefiles for examples of the commands used to build a tool. Note that the sample tools are linked with the file Stubs.o. This file contains dummy library routines used to override standard library routines that aren't used by MPW tools, thus reducing the tools' code size.

◆ *Note:* As a matter of convenience, tools are usually kept in the {MPW}Tools folder. This allows you to invoke the tool by using its simple name instead of its full pathname. {MPW}Tools is one of the directories that the Shell automatically searches when a command name is given with a partial pathname. The Shell variable {Commands} contains a comma-separated list of directories to be searched; you can easily modify it to include additional directories.

See Chapter 8 for a general introduction to linking and for instructions on linking multilingual programs. See Chapter 10 for more detailed information on linking.

Programming for the MPW Shell

This section explains how to access the MPW Shell by calling special MPW Pascal and MPW Assembler libraries. In the case of MPW C, the Shell can be accessed by using routines in the Standard C Library.

Accessing the MPW Shell—MPW C

To access the MPW 3.0 Shell environment by using MPW C, do the following:

■ Include the necessary header files

■ Link your program with CRuntime.o, CInterface.o, and Interface.o. Also link with ToolLib.o if you are using the cursor control or error management routines described in Appendix F. You may also need to link with StdCLib.o.

The standard C Library interface files contain most of the interfaces needed for programming the MPW Shell. In addition to the Standard C Library functions, MPW C contains:

■ Signal.h, containing routines that give you access to MPW software interrupts

- CursorCtl.h, containing routines to control the form and action of the cursor (see Appendix F)

- ErrMgr.h, containing routines to access messages in the Macintosh system error message file (see Appendix F)

The code for Signal.h is in CRuntime.o. The code for CursorCtl.h and ErrMgr.h is in ToolLibs.o. All interface files are in {CIncludes}.

◆ *Note:* There is an example of a C tool that runs under the MPW environment in the folder {CExamples}.

Accessing the MPW Shell—MPW Pascal

To access the MPW 3.0 Shell environment by using MPW Pascal, do the following:

- Include the statement

  ```
  USES {$U PasLibIntf.p} PasLibIntf,{$U IntEnv.p},IntEnv
  ```

  in your source text. The USES clause and the $U Compiler directive are described in the *MPW 3.0 Pascal Reference.*

- Link your program with the files Runtime.o, PasLib.o, and Interface.o. If you are using cursor-control or error message routines, you'll need to link with ToolLibs.o. (See Appendix F for information on these routines).

MPW Pascal 3.0 includes four interface files containing facilities for programs that work with the MPW Shell environment. They are

- IntEnv.p, containing the routines and data structures required to access MPW command-line parameters, Shell variables, and the standard diagnostic variable

- Signal.p, containing routines that give you access to MPW software interrupts

- CursorCtl.p, containing routines that let you control the form and action of the cursor. See Appendix F of this reference for detailed information on these routines.

- ErrMgr.p, containing routines that let you access messages in the Macintosh system error message file. See Appendix F of this reference for detailed information on these routines.

The code for IntEnv.p and Signal.p is in the library Runtime.o. The code for CursorCtl.p and ErrMgr.p is in the library ToolLibs.o.

Programmers writing tools may need to use the special facilities implemented by these interface files. They are all located in the directory {PInterfaces}.

◆ *Note:* There is an example of a Pascal tool that runs under the MPW environment, using IntEnv, in the folder {PExamples}.

Accessing the MPW Shell—Assembler

To access the MPW Shell environment from MPW Assembly language, you must do the following:

■ Import the names of the routines you are using.

■ Use the correct calling conventions.

■ Call the _RTInit function early in your program and the exit or abort procedure at the end of the program.

■ Link your assembly with the library or libraries that contain the routines' code.

These requirements are discussed in the following sections.

◆ *Note:* There is an example of an Assembly language tool that runs under the MPW environment in the folder {AExamples}.

Importing the routines

Import the names of the routines described in this appendix by using IMPORT directives. For the Shell environment and signal-handling routines, you can simply include the files IntEnv.a and Signal.a, respectively; they contain the required directives. For the cursor-control and error file management routines, you must write your own IMPORT directives in your source text.

The Shell environment and signal-handling routines are mostly C routines; hence their names are case sensitive. The cursor-control and error file management routines are all Pascal routines. Their names are not case sensitive unless CASE OBJ or CASE ON is in effect, in which case their names must be imported in capital letters.

Assembler calling conventions

Each routine described in this chapter indicates whether to use Pascal or C calling conventions.

If the calling convention is C, then push the parameters on the stack from right to left. When the function returns, its arguments will still be on the stack and its return value will be in register D0.

If the calling convention is Pascal, you must reserve space on the stack for the return value, if any. Then push the arguments from left to right. When the routine returns, the arguments will no longer be on the stack; also, the return value (if the routine was a function) will be on top of the stack.

All C functions described in this chapter leave their results in register D0. All Pascal functions described in this chapter leave their result on the stack.

The RTInit function

```
longint _RTInit (ptr retPC, longint * pargc, longint * pargv, longint * penvp
        longint forPascal)
```

One of the first calls in your program must be to the _RTInit function; the very last call should be to the exit or _exit procedure, which calls the _RTExit procedure. _RTInit is described in this section; _RTExit is described later under "Shell Utility Routines."

The _RTInit function allocates approximately 500 bytes of nonrelocatable space in the heap and calls _DataInit, the routine that initializes global data. _RTInit must be called before any of the other routines described in this section; if possible, it should be called before other code segments have been loaded.

The _RTInit function has these parameters:

■ retPC is the address to which program control should pass upon execution of _RTExit, as described under "Shell Utility Routines."

■ pArgC points to a long integer that _RTInit will set to the value of the Shell variable argc, which is discussed under "Accessing MPW Command-Line Parameters."

■ pArgV points to a pointer variable that _RTInit will set to the value of the Shell variable argv. The variable argv is discussed under "Accessing MPW Command-Line Parameters."

■ pEnvP points to a pointer variable that _RTInit will set to the vector of exported Shell variables.

■ forPascal is a numeric value passed to _RTInit. Its value should be 0 if you want the strings pointed to by envp and argv to be in C format (terminated by a zero character), and one if you want them to be in Pascal format (preceded by a length byte).

The _RTInit function returns a value of 1 if your program is being launched by the Macintosh Finder, and 0 if it is being launched by the MPW Shell. This is the value placed in the standAlone variable, described below under "Shell Utility Routines."

The function _RTInit uses C calling conventions.

For an example of the use of the _RTInit function in the code of an MPW tool, see Count.a. The routine Init shows how to call _RTInit. The exiting routine is called Stop; it shows how to call the very last call, exit.

Files to link with

The code for the Shell environment and signal-handling routines is in the library Runtime.o, except for the code for the IEGetEnv function, which is in PasLib.o. The code for the cursor-control and error file management routines is in the library ToolLibs.o. You must link the appropriate file or files to your object files if you use any of these routines.

Parameters

Parameters are passed to tools by the Shell. Every tool is passed at least one parameter: the name of the tool itself. This parameter is always the first parameter (technically, parameter 0) and is useful for error messages or other special actions.

The text that follows the command name on the command line is first analyzed by the Shell for any special processing, such as filename generation or variable substitution. (See "How Commands Are Interpreted" in Chapter 5.) This text is then split up into individual words and placed in a convenient data structure for programmatic access.

In any MPW language, there are two variables, argc and argv.

The argument vector, argv, is a pointer to an array of string pointers. Figure 12-2 demonstrates the argv structure.

■ **Figure 12-2** Parameters in MPW C and MPW Pascal

```
C Sample.a -a Sample            Pascal Sample.p -a Sample
```
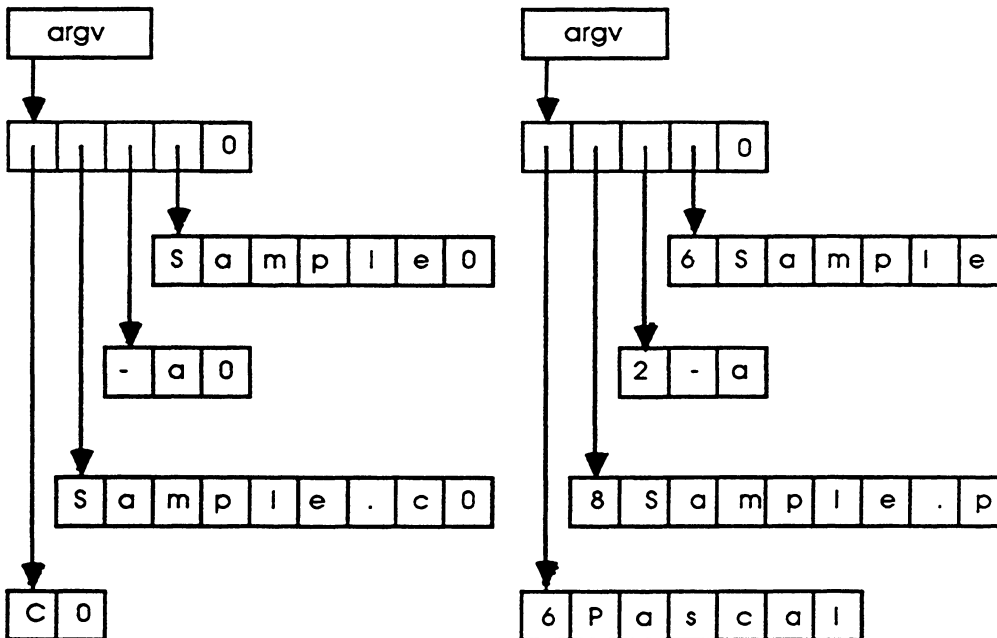

The argument count, argc, contains the number of parameters including parameter 0. The value of argc is always greater-than or equal to one, because the first parameter is always the command name. For example, in Figure 12-1, the variable argc would have the value 4.

Element 0 of argv is always the command name, as supplied by the user. When a user is running an MPW Shell script, it's important that error messages include the name of the particular MPW program that generated the error. You can include the program name with code such as this (in MPW Pascal):

```
progName := argv^[0]^;     {Store program name in temp variable.}
---
IF IOResult <> 0 THEN
    Writeln(diagnostic, progName, '-cannot open file', fileName);
```

Accessing MPW command-line parameters—MPW C

In C, the main program is actually passed three parameters, named `argc`, the argument count; `argv`, the argument vector; and `envp`, the environmental pointer. The value of `argc` includes the command name (parameter 0), and is thus always one more than the number of parameters to the command. `argv` is a pointer to a zero-terminated array of pointers to the parameters, each of which is in C string (zero-terminated) format. (See Figure 12-1.)

Accessing MPW command-line parameters—MPW Pascal

In MPW Pascal, the parameters are accessible as the unit global variables `ArgC` and `ArgV` from the IntEnv (Integrated Environment) unit. As in C, the value of `ArgC` is one more than the parameter count; `ArgV` is a pointer to a null-terminated array of Pascal string pointers.

The Integrated Environment library uses the following types and variables to allow you to access the information given in an MPW command line.

The unit IntEnv in the interface file IntEnv.p declares these types and variables:

```
TYPE
  IEString = STRING;
  IEStringPtr = ^IEString;
  IEStringVec = ARRAY [0..8191] OF IEStringPtr;
  IEStringVecPtr = ^IEStringVec;

VAR
  ArgC: LONGINT;
  ArgV: IEStringVecPtr;
  EnvP: IEStringVecPtr;
  Diagnostic: TEXT;
```

The `ArgV` variable is a pointer to an array of type `ARRAY[0..ArgC]` of Pascal string pointers, dynamically allocated and initialized by the MPW Shell when a program begins execution. Each parameter to the program is stored as a string of type `IEString` and is pointed to by a pointer in the array.

The code within the library routines creates strings of type `IEString` that are exactly the length of the arguments passed to them. For this reason, you cannot assign values to variables of type `IEString`—their values are passed directly from the MPW Shell.

Accessing MPW command-line parameters—Assembler

The Integrated Environment routine, _RTInit, can be used to access the command parameters in assembly language. The addresses of the variables argv and argc are passed to _RTInit, which initializes them.

The argv variable, set by _RTInit, is a pointer to an array of type ARRAY[0..argc] of pointers, dynamically allocated and initialized by the MPW Shell when a program begins execution. Each command-line parameter to the program is stored as a Pascal-formatted or C-formatted string (depending on the value of the forPascal parameter passed to _RTInit), pointed to by a pointer in the array.

Standard I/O channels

Before starting a tool, the Shell sets up three text I/O channels that the tool can use to communicate with the outside world. These are

- standard input

- standard output

- diagnostic output (standard error)

By default, these channels are connected to the console (that is, the frontmost, active window). Program input may be typed (or selected) and entered in any window; program output appears immediately after the command in the same window. This input and output may be taken from or directed to other files by specifying I/O redirection (<, >, >>, ≥, ≥≥, Σ, or ΣΣ) on the command line. When the Shell encounters the I/O redirection notation, it opens or creates the necessary files, removes the redirection notation from the command line so that it doesn't appear in the program's parameter list, and then arranges for the open files to be passed to the program. When the tool finishes, the Shell flushes any buffered output and closes the files.

I/O buffering

When using I/O routines provided by the language libraries, varying degrees of buffering are expected to occur on the standard I/O channels:

- Input from the console is buffered until the Enter key is pressed. If there is a selection when Enter is pressed, the selected text is used to satisfy the console read request; otherwise, the entire line that contains the insertion point is given to the reader.

  ◆ *Note:* The MPW method of reading input creates a difficulty for interactive tools that write prompting text and pause to read a response entered on the same line: The tool will receive the prompt back as part of the line read, unless there was a selection when Enter was pressed.

- When input is taken from a file, the I/O package will, by default, read the data from the disk in 1K blocks.

- Text written to standard output is also buffered 1K at a time before being sent to a file or to the console. (As a convenience, when a read request is issued to the console, all interactive output buffers are flushed so that any prompting text will appear before the program pauses waiting for input.)

- Text written to the diagnostic channel is buffered one line at a time, so that error messages and progress information appear in a timely manner while the program is executing.

Note that this buffering can cause apparently anomalous behavior: In particular, if both standard output and diagnostic output are directed to the console, the order of the output on the screen may not match the order in which the data was written.This change in order may result because the separate buffers are flushed at different times, as illustrated in Figure 12-3. You can circumvent this problem by flushing standard output before writing to diagnostic output.

◆ *Note:* Figure 12-3 shows the output conventions in C and Pascal. Assembly-language programmers must do their own buffering, or call C or Pascal routines.

■ **Figure 12-3** I/O buffering
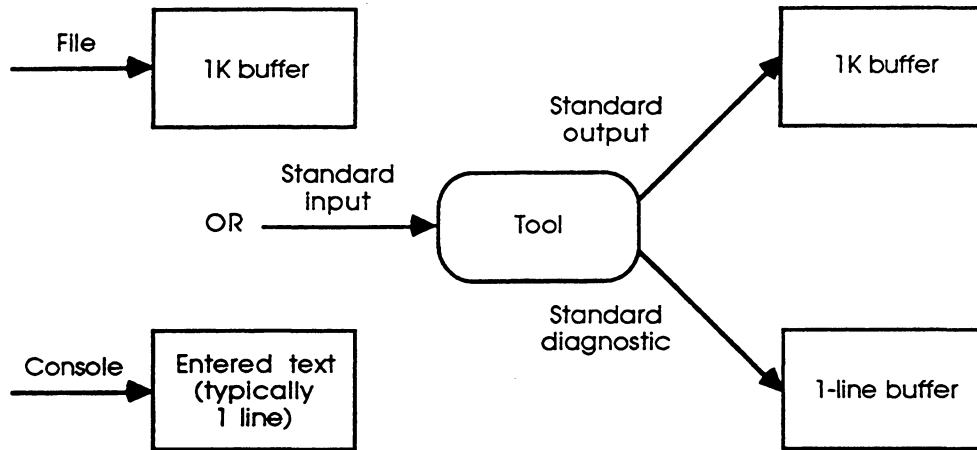
C   The standard I/O files are available for reading or writing in C, via the file descriptors 0, 1, and 2, or the StdIO stream descriptors `stdin`, `stdout`, `stderr`. These descriptors are fully documented in the *MPW 3.0 C Reference.*

Pascal   In Pascal, the program parameters `Input` and `Output` correspond to the standard input and output channels. A text file variable called `diagnostic`, which is connected to the standard diagnostic channel, is available from the IntEnv unit. Most tools written in Pascal can use the standard Pascal input and output functions with the text files `Input`, `Output`, or `Diagnostic`. The use of these parameters is documented in detail in the *MPW 3.0 Pascal Reference.*

I/O to windows and selections

The MPW environment also provides to tools the ability to read and write to windows or to selections within windows. No special programming is required to use this feature. The MPW Shell monitors file system calls, and intercepts those that refer to a file that is currently open as a window. These calls are redirected automatically to the window rather than the file. (Thus, any modifications to the file do not become permanent until the window is saved.)

Accessing *selections* within windows is equally transparent to programs. All that is required is that the filename contain the selection suffix (.§). Reading from a selection is the same as reading from a file, and the beginning and end of the selection are treated as the bounds of the file. However, writing to a selection *replaces* the selection and has the interesting property that the data written is inserted into the file, rather than overwriting the data that follows.

Because window and selection I/O is handled automatically by the MPW Shell, tools should simply assume that they are always dealing with files.

Error information

All Shell I/O routines report errors by setting the value of the integer variable `errno`. In addition, the routines `open`, `close`, `read`, `write`, and `ioctl` set the variable `MacOSErr`. The error values are shown in Table 12-1.

MPW C The variables `errno` and `MacOSErr` are global variables.

MPW Pascal Results are reported with `IOresult`, which looks at both `errno` and `MacOSErr`. If `IOresult` is positive, it holds `errno`. If `IOresult` is negative, it holds `MacOSErr`.

MPW Assembly `IMPORT` the variables `errno` (a long) and `MacOSErr` (a word). You can import these variables with the IntEnv.a interface file.

The variable `errno` is an integer. Its behavior is described in the *MPW 3.0 C Reference*. The values of `errno` are typically small positive integers. Zero means that there is no error. However, libraries do *not* set `errno` to zero on successful calls.

`MacOSErr` is a short that holds the error result from Macintosh toolbox calls made by the libraries (such as the result of a file system call made by the `ioctl` function). `MacOSErr` holds zero if there is no error; if it holds a negative number, that means there is an error. See *Inside Macintosh* for details on error numbers.

■ **Table 12-1** Shell I/O errors

Value	Identifier	Message	Explanation
2	ENOENT	No such file or directory.	This error occurs when a file whose filename is specified does not exist or when one of the directories in a pathname does not exist.
3	ENORSRC	Resource not found	A required resource was not found. This error applies to `faccess` calls that return tab, font, or print record information.
5	EIO	I/O error	Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.
6	ENXIO	No such device or address	I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.
7	E2BIG	Insufficient space for return argument	The data to be returned is too largefor the space allocated to receive it.
9	EBADF	Bad file number	Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).
12	ENOMEM	Not enough space	The system ran out of memory while the library call was executing.
13	EACCES	Permission denied	An attempt was made to access a file in a way forbidden by the protection system.

(Continued)

Value	Identifier	Message	Explanation
17	EEXIST	File exists	An existing file was mentioned in an inappropriate context.
19	ENODEV	No such device	An attempt was made to apply an inappropriate system call to a device; for example, to read a write-only device.
20	ENOTDIR	Not a directory	An object that is not a directory was specified where a directory is required; for example, in a path prefix.
21	EISDIR	Is a directory	An attempt was made to write on a directory.
22	EINVAL	Invalid parameter	Some invalid parameter was provided to a library function.
23	ENFILE	File table overflow	The system's table of open files is full, so temporarily a call to open cannot be accepted.
24	EMFILE	Too many open files	The system cannot allocate memory to record another open file.
28	ENOSPC	No space left on device	During a write to an ordinary file, there is no free space left on the device.
29	ESPIPE	Illegal seek	An lseek was issued incorrectly.
30	EROFS	Read-only file system	An attempt to modify a file or directory was made on a device mounted for read-only access.
31	EMLINK	Too many links	An attempt to delete an open file was made.

Shell I/O routines—MPW C

The MPW C input and output routines are part of the comprehensive Standard C Library. The Standard C Library is a collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform mathematical operations. You may use any of the Standard C Library routines or low-level routines individually described later in this chapter. For more information, see the *MPW 3.0 C Reference.*

stdio—standard buffered input/output package

The Standard I/O package constitutes an efficient user-level I/O buffering scheme. The inline macros `getc` and `putc` handle characters quickly.

The following macros and higher-level routines all use `getc` and `putc`:

getchar	putchar	fgetc	fgets
fprintf	fputc	fputs	fread
fscanf	fwrite	gets	getw
printf	puts	putw	scanf

Calls to these macros and functions can be freely intermixed.

The constants and the following functions are implemented as macros:

getc	getchar	putc	putchar
feof	ferror	clearerr	fileno

Avoid redeclaration of these names.

Any program that uses the Standard I/O package must include the <StdIO.h> header file of macro definitions. The functions, macros, and constants used in the Standard I/O package are declared in the header file and need no further declaration.

A *stream* is a file with associated buffering and is declared to be a pointer to a `FILE` variable. Functions `fopen`, `freopen`, and `fdopen` return this pointer. The information in the `FILE` variable includes

■ the file access—read or write

■ the file descriptor as returned by `open`, `creat`, `dup`, or `fcntl`

■ the buffer size and location

■ the buffer style (unbuffered, line buffered, or file buffered)

Output streams, with the exception of the standard error stream `stderr`, are by default file buffered if the output refers to a file. File `stderr` is by default line buffered. When an output stream is **unbuffered**, it is queued for writing on the destination file or window as soon as written; when it is **file buffered**, many characters are saved up and written as a block; when it is **line buffered**, each line of output is queued for writing as soon as the line is completed (that is, as soon as a newline character is written). Function `setvbuf` may be used to change the stream's buffering strategy.

Normally, there are three open streams with constant pointers declared in the <StdIO.h> header file and associated with the standard open files:

■ **Table 12-2**   Standard files

FILE variable	Fildes	Description	Buffer style
stdin	0	standard input file	file buffered
stdout	1	standard output file	file buffered
stderr	2	standard error file	line buffered

Buffer initialization: The `FILE` variable returned by `fopen`, `freopen`, or `fdopen` has an initial buffer size of 0 and a `NULL` buffer pointer. The buffer size is set and the buffer allocated by a call to `setbuf`, `setvbuf`, or the first I/O operation on the stream, whichever comes first. Buffer initialization is done using the following algorithm:

1. If `_IONBF` (no buffering) was set by a call to `setvbuf`, initialization steps 2 and 3 are skipped. The buffer size remains 0 and the buffer pointer remains `NULL`.

2. Checks the access-mode word for `_IOLBF` (line buffering). This bit is usually set only in the predefined `FILE` `stderr`, but a call to `setvbuf` can set it for any file. If line buffering is set, the buffer size is set to `LBUFSIZ` (100). If line buffering is not set, `ioctl` is called with an `FIOBUFSIZE` request and the buffer size is set to the returned value or to `BUFSIZ` (1024) if no value is returned.

3. If the buffer pointer is `NULL`, a request is made for a buffer whose size was determined in step 2; the buffer pointer is set to point to the newly allocated buffer. If the requested size cannot be allocated, attempts are made to allocate `BUFSIZ` or `LBUFSIZ` if these are smaller than the requested size. If all requests fail, the buffer pointer remains `NULL` and the `_IONBF` (no buffering) bit is set.

4. Function `ioctl` is called with an `FIOINTERACTIVE` request; if it returns `true`, the `_IOSYNC` bit is set in the access-mode word. This is done for all `FILE` variables, regardless of their buffering style and size. (The `_IOSYNC` bit is described in the next section.)

The `setvbuf` function lets you specify values for buffer size, buffer pointer, and access-mode word other than the default values of 0, `NULL`, and 0, respectively. The `setvbuf` function must be called before the first I/O operation occurs, so that the buffer initialization procedure described above receives the values you specify instead of the default values.

Buffered I/O: On each write request, the bytes are transferred to the buffer and an internal counter is set to account for the number of bytes in the buffer. If `_IOLBF` is set and a newline character is encountered while transferring bytes to the buffer, the buffer is flushed (written immediately) and the transfer continues at the beginning of the buffer. This continues until the write-request count is satisfied or a write error occurs.

On each read request, the `_IOSYNC` bit in the access-mode word is checked. If `_IOSYNC` is on, all current `FILE` variables that have `_IOSYNC` on and are open for writing are flushed. In other words, a read from an interactive `FILE` variable flushes all interactive output files before reading. This ensures that any prompts, I/O in a window, or other visual feedback is displayed before the read is initiated. Then if the internal counter is 0, an entire buffer is read into memory if possible. (For the console device, less than a buffer's worth is likely to be read.) The bytes required to satisfy the read request are transferred, going back to the device for more if necessary, and an internal pointer is advanced if any bytes remain unread.

When the Standard I/O package is used, Standard I/O cleanup is performed just before termination of the application. Any normal return including a call to `exit` causes Standard I/O cleanup, which consists of a call to `fclose` for every open `FILE` stream.

▲ **Warning**   Do not use a file descriptor (0, 1, or 2) where a FILE variable (stdin, stdout, or stderr) is required. File <StdIO.h> includes definitions other than those described above, but their use is not recommended. Invalid stream pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions. ▲

An integer constant `EOF` (−1) is returned upon end of file or error by most integer functions that deal with streams. See the descriptions of the individual functions for details.

You may also refer to these Standard C Library routines:

close	exit	fclose	ferror
fopen	fread	fseek	getc
gets	lseek	onexit	open
printf	putc	puts	read
scanf	setbuf	ungetc	write

Shell I/O routines—MPW Pascal

The Integrated Environment library includes four general I/O routines that you can use in conjunction with the standard Pascal I/O routing from MPW Pascal programs that run within the MPW environment. These functions are listed, where available, in the next section.

Shell I/O routines—Assembler

Eight general I/O routines are available for use with MPW Assembler programs that run within the MPW environment

Shell I/O routines

In the sections that follow, each I/O routine is individually described, along with the appropriate calls in MPW C, MPW Pascal, and MPW Assembly language.

open—open for reading or writing

```
int open(char *filename, int mode)
```

The Shell routine open opens the file, window, or selection named by filename for both reading and/or writing. The parameter mode sets the file-status flags, and specifies file creation, truncation, and/or exclusive access.

To construct `mode`, first select one of the following access modes:

- `O_RDONLY`   Open for reading only.

- `O_WRONLY`   Open for writing only.

- `O_RDWR`   Open for reading and writing.

Then optionally add one or more of these modifiers:

- `O_APPEND`   The file pointer is set to the end of the file before each write.

- `O_CREAT`   If the file does not exist, it is created.

- `O_TRUNC`   If the file exists, its length is truncated to 0; the mode is unchanged.

- `O_RSRC`   The file's resource fork is opened. (Normally, the data fork is opened.)

The following setting is valid only if `O_CREAT` is also specified:

- `O_EXCL`   Function `open` fails if the file exists.

Upon successful completion, a nonnegative integer (the file descriptor) is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The named file is opened unless one or more of the following are true:

- `O_CREAT` is not set and the named file does not exist. [`ENOENT`]

- More than about 30 file descriptors are currently open. The actual limit varies according to runtime conditions. [`EMFILE`]

- `O_CREAT` and `O_EXCL` are set and the named file exists. [`EEXIST`]

MPW C

```
int open(char *filename, int mode)
```

MPW Pascal

```
PROCEDURE IEopen(VAR fvar: univ PascalFile; filename: string; mode:
longint);
```

After `IEopen` executes successfully, `fvar` contains a pointer to the beginning of the file named by *filename*.

Normally, MPW Pascal tools will use the built-in calls `Reset`, `Rewrite`, or `Open`. The Procedure `IEopen` provides additional options with the `mode` parameter. After using `IEopen`, the tool should then use the built-in MPW Pascal calls `Read`, `Write`, and `Close` using the `fvar` variable.

MPW Assembler

```
longint open(char *filename, longint mode)
```

Use the C routine `open`. After `open` executes successfully, D0 contains an integer file descriptor (a nonnegative integer), with the file pointer set to the beginning of the file. File descriptors for input, output, and the diagnostic output are predeclared in the `include` file IntEnv.a, as shown in Table 12-3.

- **Table 12-3**   Predeclared file descriptors

Value	Identifier	File
0	InputFD	Standard input
1	OutputFD	Standard output
2	DiagnosticFD	Diagnostic output

If there is an error, D0 will contain -1 and `errno` will be set to indicate the error.

close—close a file descriptor

```
int close(int fd)
```

The close function closes the file associated with the file descriptor fd. (The file descriptor is obtained from an `open` call.)

Function `close` fails if `fd` is not a valid open file descriptor.

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

MPW C

```
int close(int fd)
```

(File descriptor `fd` may also be obtained from a `creat`, `dup`, or `fcntl` call.)

MPW Pascal

To close a file opened with `IEopen`, use the MPW Pascal built-in procedure `Close`.

MPW Assembler

```
longint close(longint fd)
```

Use the C function `close`. If successful, close sets D0 to 0.

read—read from a file

```
int read(int fd, char *buf, unsigned nbyte)
```

On devices capable of seeking, read starts reading at the current position of the file pointer associated with fd. Nonseeking devices always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon return from read, the file pointer is incremented by the number of bytes actually read.

File descriptor fd is obtained from a call to open. Function read transfers up to nbyte bytes from the file associated with fd into the buffer pointed to by buf.

Upon successful completion, read returns the number of bytes actually read and placed in the buffer; this number may be less than nbyte if the file is associated with a window or if the number of bytes left in the file is less than nbyte bytes.

File descriptor 0 is opened by the MPW Shell as standard input.

A value of 0 is returned when an end of file has been reached, or –1 if a read error occurred. Upon successful completion, a nonnegative integer is returned indicating the number of bytes actually read. Otherwise, –1 is returned and errno is set to indicate the error.

Function read fails if fd is not a valid file descriptor open for reading.

MPW C

```
int read(int fd, char *buf, unsigned nbyte)
```

(File descriptor fd may also be obtained from a creat, dup, or fcntl call.)

MPW Pascal

To read from a file opened with IEOpen, use the MPW Pascal built-in procedure Read.

MPW Assembler

```
longint read(longint fd, char *buf, unsigned longint nbyte)
```

Use the C routine read. If successful, read leaves the number of bytes actually read in D0 (which may be less than nbyte, if the end-of-file was encountered); otherwise it sets D0 to –1 and sets the value of errno.

write—write to a file

```
int write(int fd, char *buf, unsigned nbyte)
```

The function write attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the fd. (File descriptor fd is obtained from an open.) Internal limitations may cause write to write fewer bytes than requested; the number of bytes actually written is indicated by the return value. Several calls to write may therefore be necessary to write out the contents of buf.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from write, the file pointer is incremented by the number of bytes actually written. On nonseeking devices, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the o_APPEND file status flag set in open is on, the file pointer is set to end of file before each write.

The file pointer remains unchanged and write fails if fd is not a valid file descriptor open for writing.

If you try to write more bytes than there is room for on the device, write writes as many bytes as possible. For example, if nbyte is 512 and there is room for 20 bytes more on the device, write writes 20 bytes and returns a value of 20. The next attempt to write a nonzero number of bytes will return an error.

File descriptor 1 is standard output; file descriptor 2 is standard error.

Upon successful completion, the number of bytes actually written is returned. Otherwise, –1 is returned and errno is set to indicate the error.

MPW C

```
int write(int fd, char *buf, unsigned nbyte)
```

(File descriptor fd may also be obtained from a creat, dup, or fcntl call.)

MPW Pascal

To write to a file opened with IEopen, use the MPW Pascal built-in procedure write.

MPW Assembler

```
longint write(longint fd, char *buf, unsigned longint nbyte)
```

Use the C routine write.

lseek—move read/write file pointer

```
int lseek(int fd, int offset, int whence)
```

The function `lseek` moves the read/write file pointer in the file associated with `fd`, according to the following value of `whence` and `offset`:

- If `whence` is 0, the pointer is set to `offset` bytes.

- If `whence` is 1, the pointer is set to its current location plus `offset`.

- If `whence` is 2, the pointer is set to the size of the file plus `offset`.

- If `whence` is 1 or 2, the value of `offset` may be negative.

Upon successful completion, the file pointer value, as measured in bytes from the beginning of the file, is returned.

The file pointer remains unchanged and `lseek` fails if one or more of the following are true:

- File descriptor `fd` is not open. [EBADF]

- Parameter `whence` is not 0, 1, or 2. [EINVAL]

- The resulting file pointer would point past end of file. [ESPIPE]

- The resulting file pointer would point before beginning of file. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined. Upon successful completion, a nonnegative long integer indicating the file-pointer value is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

> ▲ **Warning** Function `lseek` has no effect on a file opened with the `O_APPEND` flag because the next write to the file always repositions the file pointer to the end before writing. ▲

MPW C

```
int lseek(int fd, int offset, int whence)
```

MPW Pascal

```
FUNCTION IElseek(VAR fvar: UNIV PascalFile; offset: LONGINT; whence:
LONGINT): LONGINT;
```

Do not use `IELseek` with a structured file.

MPW Assembler

```
longint lseek(longint fd, longint offset, longint whence)
```

Use the C function `lseek`.

---

fcntl—file control

```
int fcntl(int fd, unsigned int cmd, int arg)
```

Function `fcntl` duplicates a file descriptor. A file remains open until all its file descriptors are closed. Parameter `fd` is an open file descriptor obtained from an `open` call. Parameter `cmd` takes the value `F_DUPFD`, which tells `fcntl` to return the lowest numbered available file descriptor greater than or equal to `arg`.

Normally `arg` is greater than or equal to 3, in order to avoid obtaining the standard file descriptors 0, 1, and 2. Function `fcntl` returns a new file descriptor that points to the same open file as `fd`. The new file descriptor has the same access mode (read, write, or read/write) and file pointer as `fd`. Any I/O operation changes the file pointer for all file descriptors that share it.

Function `fcntl` fails if one or more of the following are true:

- Parameter `fd` is not a valid open file descriptor. [EBADF]

- Parameter `arg` is negative or greater than the highest allowable file descriptor. [EINVAL]

Upon successful completion, the value returned is a new file descriptor. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

♦ *Note:* The `F_GETFD`, `F_SETFD`, `F_GETFL`, and `F_SETFL` commands of `fcntl` are not supported on the Macintosh.

MPW C

```
int fcntl(int fd, unsigned int cmd, int arg)
```

MPW Pascal

The function `fcntl` is not supported in MPW Pascal.

MPW Assembler

```
longint fcntl(longint fd, unsigned longint cmd, longint arg)
```

Use the C routine `fcntl`.

IOCtl—communicate with device handler

The function `ioctl` communicates with a file's device handler by sending control information and/or requesting status information.

The `cmd` parameter specifies one of the following device-specific operations:

FIOINTERACTIVE
: Return a value of 0 if the device is interactive, –1 otherwise. Ignore `arg`.

FIOBUFSIZE
: Return the default buffer size for the device. The buffer size is expressed in bytes and is returned as a `longint` value pointed to by `arg`. If the device has no default buffer size, `ioctl` returns a value of –1; it returns 0 otherwise.

FIOFNAME
: Store the filename associated with `fd` in a character array 255 characters in size, pointed to by `arg`. `IEIOCtl` returns a value of –1 if the filename length exceeds 255 characters, 0 otherwise.

FIOREFNUM
: Return the Macintosh file reference number associated with `fd`. The reference number is returned as an `integer` value pointed to by `arg`. `ioctl` returns a value of –1 if the file associated with `fd` is not open on a Macintosh file (such as the console device), 0 otherwise.

FIOSETEOF
: Set the logical end of the file associated with `fd` to the value of `arg`, which becomes the new size of the file in bytes. This command can be used to reduce or increase the size of an open file. The current file pointer is not affected unless the file size is set to a value lower than the position to which it points.

TIOFLUSH
: Discard unread terminal input. This parameter value is used only for the console device and other terminal devices. `ioctl` returns a value of –1 if the file associated with `fd` is not a terminal device, 0 otherwise. Parameter `arg` is ignored.

MPW C

```
int ioctl(int fd, unsigned int cmd, long *arg)
```

The `cmd` constants are in `IOCtl.h`.

MPW Pascal

```
FUNCTION IEioctl(VAR fvar: UNIV PascalFile; cmd: LONGINT; arg: UNIV
LONGINT) : LONGINT;
```

The `cmd` constants are defined in `IntEnv` unit.

MPW Assembler

```
longint ioctl(longint fd, unsigned longint cmd, longint *arg)
```

Use the C function `ioctl`. The `cmd` constants are defined in `IntEnv.a`.

Shell utility routines

These utilities are useful when writing an MPW tool. The utility routines provide methods to:

- determine whether a program is running under the MPW Shell (`StandAlone`)
- to access the values of MPW Shell variables (`getenv`)
- to specify exit handlers (`atexit`)
- to terminate the current application (`exit`)
- to access information about MPW Shell documents (`faccess`)

StandAlone—check whether running under the MPW Shell

The standard libraries provide a method to tell whether a program is running under the MPW Shell.

MPW C

The global variable `StandAlone` is an int. If `StandAlone` is zero, the program is running under the MPW Shell.

MPW Pascal

```
FUNCTION IEStandalone: BOOLEAN;
```

The `IEStandalone` function returns a result of type `boolean`. The result is `false` if the program is running under MPW, `true` if it is not.

MPW Assembler

Import the longint variable `StandAlone` (in the Interface file IntEnv.a). If `StandAlone` is non-zero, the program is running under the MPW Shell.

getenv—access exported MPW Shell variables

```
char *getenv(char *varname)
```

The MPW Shell maintains a set of state variables that can be made available to tools with the Export command. (See "Variables" in Chapter 5 for the list of standard exported Shell variables.) Whenever you run a tool, the Shell makes a copy of the names and string values of all exported variables and passes this list to the program. The tool can then determine the value of a variable by one of two methods:

- doing a linear search of the list of variables until the desired variable name is found

- using the getenv function

Because only a copy is passed, a tool cannot alter the Shell's value of a variable.

Function getenv searches the environment for a Shell variable with the name specified by varname and returns a pointer to the character string containing its value. The null pointer is returned if the Shell variable is not defined or has not been exported. The Shell-variable name search is case-insensitive.

For standalone applications, which do not run under the MPW Shell, getenv always returns the null pointer.

MPW C

```
char *getenv(char *varname)
```

The environment can also be accessed by means of a parameter to the C main-entry-point function main if the main procedure is declared as
```
main(argc, argv, envp)
```

The envp array represents the set of MPW Shell variables that have been made available to tools by means of the MPW Export command. The ith envp entry has the form
```
envp[i] = "varname\0varvalue\0";
```

The last envp entry is the null pointer.

If you use envp to search the environment, be sure to use case-insensitive string comparisons.

MPW Pascal

```
FUNCTION IEgetenv(envName: STRING; VAR envValue: UNIV IEString):
BOOLEAN;
```

IEGetEnv returns TRUE if it is successful in finding the value of a variable that is defined and exported in the MPW Shell environment. The parameter envName is a Pascal string naming an exported Shell variable, with uppercase and lowercase not distinguished. The parameter envValue is returned with the value of the Shell variable. IEGetEnv returns FALSE if it cannot find the variable.

Pascal programmers are also provided with another IntEnv unit global variable, called EnvP. The variable EnvP points to a list of variable name and value pairs. The structure used is the same as that for C, except that the varname is in Pascal string format.

◆ *Note:* VarValue is in c string format, that is, null-terminated with no length-byte. Please refer to Figure 12-4.

■ **Figure 12-4**   Format of envp array for MPW C and MPW Pascal
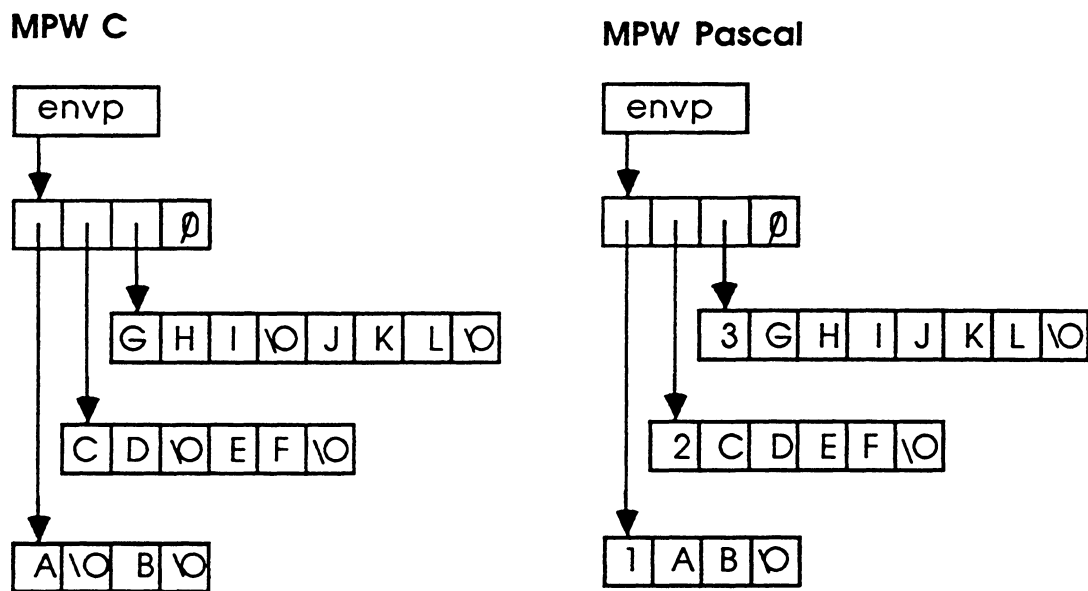
MPW Assembler

```
char *getenv(char *varname)
Function IEgetenv(envName: string; VAR envValue: univ IEString):
boolean;
```

Use getenv if the value of the forPascal parameter in the _RTInit call was 0; otherwise use IEGetEnv.

The Integrated Environment routine, _RTInit, can also be used to access Shell variables in assembly language. The address of envp is passed to _RTInit, which initializes it. You can choose Pascal or C strings (by setting forPascal to the appropriate value in the call to _RTInit).

▲ **Warning**   Functions getenv and IEGetEnv return a pointer to the place in memory where a copy of the MPW Shell variable resides. Do not modify the value of a Shell variable in such a way as to increase its length. ▲

---

atexit—install a function to be executed at program termination

```
int atexit(void (*func)(void))
```

Normal program termination closes and flushes open files and releases program memory. If you want additional exit processing, you can use atexit to insert a routine that is executed just before normal termination. The parameter func is a pointer to such a routine. Up to 32 exit procedures are permitted (not including the one used by the Standard I/O Package to flush all the buffers). The routines specified will be executed in the reverse order of their installation. The routines will be called with no parameters.

MPW C

```
int atexit(void (*func)(void))
```

The routine atexit returns a zero value if the installation succeeds.

MPW Pascal

```
PROCEDURE IEatexit(exitProc: UNIV LONGINT); C;
```

The exit routines cannot be nested procedures.

MPW Assembler

```
int atexit(void (*func)(void))
```

Use the MPW C atexit routine.

▲ **Warning**   If a function is installed more than once, it will be executed as many times as it was installed. ▲

exit—terminate the current application

```
void exit(int status)
void abort()
```

The functions `exit` and `abort` close open file descriptors and terminate the application or tool. Here is the order in which `exit` performs its duties:

1. It executes all exit procedures in reverse order of their installation by `atexit`, followed by the exit procedures for the Standard I/O package if Standard I/O routines were used. All buffered files are flushed and closed.

2. It closes all open files that were opened with `open`.

3. If the program is a tool running under the MPW Shell, `exit` places the lower three bytes of `status` into the Shell's `status` variable and returns control to the MPW Shell.

4. If the program is an application, `exit` terminates the application.

There is no return from `exit` or `abort`.

The functions `exit` and `abort` do not close files your tool opened with calls to the I/O routines documented in *Inside Macintosh*. However, the MPW Shell closes them after the tool returns.

Status should be 0 for normal execution or a small positive value for errors. (See the section "Status Codes" at the beginning of this chapter.)

The function `exit` takes a value that will be returned to the caller; `abort` does not.

MPW C

```
void exit(int status)
void abort()
```

Notice that in MPW C the `main` program is a function that returns an integer. The return value of `main` is interpreted by the MPW Shell as the program status. `Main` programs that return to the Shell without setting `status` to an integer value will return a random status.

MPW Pascal

```
PROCEDURE IEexit(status: LONGINT); C;
PROCEDURE IEabort(); C;
```

MPW Assembler

```
void exit(longint status)
void abort()
```

Both the `exit` and `abort` procedures terminate a program running under the MPW Shell by calling `_RTExit`. The action of `_RTExit` is described below.

```
_RTExit(longint status);
```

The _RTExit procedure must be the last executed routine in a tool running under the MPW Shell. It calls any routines installed by the atexit procedure (described above) and then returns control to the address specified by the retPC parameter in the original _RTInit call.

Programs normally call the exit or abort procedure, described above.

faccess—named file access and control

```
int faccess(char *filename, unsigned int cmd, long *arg)
```

The function faccess provides access to control and status information for named files.

The parameter cmd must be set to one of the constants in the following list to indicate what operation is to be performed on the file. As noted in the list, some calls to faccess also require the arg parameter, usually as a long or as a pointer to a long.

The following commands are available to all programs:

F_DELETE   Deletes the named file, or returns an error if the file is open or in a window. Arg is ignored.

F_RENAME   Renames the named file. Arg is a pointer to a string containing the new name.

The following commands are available to programs running under the MPW Shell. All of these calls can be used on open or closed files.

F_GTABINFO   Returns the tab setting for an MPW text file named by filename. Arg is a pointer to a long integer. The long integer's value is the tab setting expressed as the number of spaces in the text file's font.

F_STABINFO   Sets the tab setting for an MPW text file named by filename. Arg is a long integer representing the tab setting expressed as the number of spaces in its font.

F_GFONTINFO   Returns the font and font size of an MPW text file named by filename. Arg is a pointer to a long integer. The font number is stored in the upper word of the long integer; the font size is stored in the lower word.

F_SFONTINFO   Sets the font and font size of an MPW text file named by
               filename. Arg is a long integer. The font number is read from the
               upper word of the long integer; the font size is read from the lower
               word.

F_GPRINTREC   Gets a print record TPrint for the MPW text file filename.
               Arg is a handle to the print record. Before calling faccess with
               this cmd value, the Macintosh Printing Manager must be initialized
               and the print record handle THPrint must be allocated.

F_SPRINTREC   Sets a print record for the MPW text file filename. Arg is a
               handle to the print record. Before calling faccess with this cmd
               value, the Macintosh Printing Manager must be initialized and the
               print record handle THPrint must be allocated.

F_GSELINFO   Gets the selection information for the MPW text file filename.
               Arg is a pointer to a selection record.

               A selection record is a C structure (or Pascal record) in this form:

```
struct SelectionRecord {
        long startingPos;
        long endingPos;
        long dispayTop
};
```

               The startingPos is the starting position of the selection, the
               endingPos is the ending position of the selection, and displayTop
               is the position of the first character at the top of the window. All
               three positions are offsets from the beginning of the file, with the
               first position in the file being 0.

F_SSELINFO   Sets the selection information for the MPW text file filename.
               Arg is a pointer to a selection record described above. The display
               will start on the line that contains the character displayTop.
               DisplayTop does not have to be the first character in a line. The
               window will not automatically scroll horizontally to display the
               actual character specified. It is invalid to set startingPos less than
               zero, greater than endingPos, or greater than the length of the file.
               It is also invalid to set displayTop to a value greater than the length
               of the file. If displayTop is negative, it will be ignored, and only
               startingPos and endingPos will be used. (This is useful if you want
               the MPW Shell to provide for scrolling only when necessary. If
               displayTop is greater than 0, scrolling will be done on each
               faccess call.)

F_GWININFO	Gets the current window position. Arg is a pointer to a rectangle (of type Rect) to store the information. The rectangle is in global coordinates.
F_SWININFO	Sets the current window position. Arg is a pointer to a rectangle (of type Rect) specifying the new size and position. If the window size is invalid, or the rectangle is completely off the screen, faccess returns –1.
F_OPEN	Reserved for operating system use.

If faccess is successful it returns a nonnegative value, usually 0. If the file cannot be accessed, faccess returns –1. If the requested resource for F_GTABINFO, F_GFONTINFO, or F_GPRINTREC does not exist for the named file, default values are stored and the function returns a value greater than 0.

MPW C

```
int faccess(char *filename, unsigned int cmd, long *arg)
```

The cmd constants are declared in the file FCntl.h. If faccess returns with an error, it also sets the value of errno.

MPW Pascal

```
FUNCTION IEfaccess(filename: STRING; cmd: LONGINT; arg: UNIV LONGINT):
LONGINT;
```

The cmd constants are declared in the unit IntEnv. All strings are Pascal strings.

MPW Assembler

```
longint faccess(char *filename, unsigned longint cmd, longint *arg)
```

Use the C function faccess. All strings are C strings. The cmd constants are declared in the file IntEnv.a. If faccess returns with an error, it also sets the value of errno.

Signal handling

The MPW environment provides a set of routines to handle signals. A **signal** is similar to a hardware interrupt in that its invocation can cause program control to be temporarily diverted from its normal execution sequence; the difference is that the events that raise a signal reflect a change in program state rather than hardware state. Examples of signal events are stack overflow, heap overflow, software floating-point exceptions, and Command-period interrupts.

Signal handling is available only for tools that run under the MPW Shell; it is not available for applications that run under the Macintosh Finder.

◆ *Note:* There are just two software interrupts that can be detected by a program running under the current version of the MPW Shell. One is the Command-period, represented by the value `SIGINT`. The other is abnormal termination by the Abort function, represented by the value `SIGABRT`. As additional software interrupts are added, new values will be added to represent them. The signal-handling procedures will then accept these new values.

The default action of any signal is to close all open files, execute any exit procedures (described above under "`exit`"), and terminate the program. If, however, your tool requires special handling of a signal, or chooses to ignore it, you can use the procedure `signal` to replace the default signal handling procedure with your own procedure.

Signal handling—C

To access the signal handler in MPW C, do the following:

■ Include the file Signal.h in your source text.

■ Link your program with the file CRuntime.o.

◆ *Note:* The type definition `signalHandler`, used later in this section, is *not* included in the file Signal.h. `signalHandler` is equivalent to:

```
Typedef void (*SignalHandler) (int);
```

Signal handling—Pascal

To access the signal handler in MPW Pascal, do the following:

■ Include the statement

```
USES {$U Signal.p} Signal
```

in your source text. The USES clause and the $U Compiler directive are described in the *MPW 3.0 Pascal Reference.*

■ Link your compilation with the files Runtime.o and PasLib.o.

The unit Signal declares the following types:

```
SignalMap = integer;
SignalHandler = ^longint;
```

Signal handling—Assembler

To access the signal handler, do the following:

■ Include the file Signal.a in your source text.

■ Link your program with the file Runtime.o.

Signal—specify a signal handler

```
void (*signal (int signum, void (*newHandler)(int)))(int);
```

Function `signal` replaces the current signal handler (the procedure to be executed upon receipt of the signal specified in `signum`) with a user-supplied signal handler. The default signal handler may be set or restored by specifying `SIG_DFL` as the current signal handler.

Some predefined signal handlers may be specifed as the `newHandler`. The function `SIG_IGN` does nothing. It may be used as the `newHandler` in a call to `signal` to ignore the signal. The function `SIG_DFL` is the default signal handler. It calls the program's exit procedure.

The `newHandler` function that is passed to `signal` takes one parameter (a long integer). The parameter is the number of the signal that is currently being handled. Writing a signal handler is described below.

Function `signal` returns the previous `SignalHandler` pointer. If this pointer must be restored in another part of the program, save the return value and restore it with another call to `signal`.

MPW C

```
void (*signal (int signum, void (*newHandler)(int))) (int);
```

Alternatively, you can use the equivalent:

```
Typedef void (*SignalHandler)(int);
SignalHandler *signal(int signum, SignalHandler *newHandler)
```

MPW Pascal

```
FUNCTION IEsignal(SigNum: LONGINT; SigHdlr: UNIV SignalHandler):
 SignalHandler; C;
```

MPW Assembler

```
signalhandler *signal(longint signum, SignalHandler *newHandler)
```

Use the C function `signal`.

Raise—raise a signal

```
int raise(int signum)
```

The `raise` allows signals to be raised under program control. It sends the signal `signum` to the program. It returns 0 if successful, nonzero otherwise. Notice that depending on the signal handler installed, `raise` might not return.

MPW C

```
int raise (int signum)
```

MPW Pascal

```
FUNCTION IEraise(SIGNUM:LONGINT):LONGINT; C;
```

MPW Assembler

```
longint raise(longint signum)
```

Use the C function `raise`.

Writing a signal handler

```
void signalHandler(int signum)
```

When a signal is raised, a call is made to the handler specified as the parameter `newHandler` in a call to `signal`. One parameter is passed to the signal handler. This parameter, `signum`, is the signal number currently being handled.

When the tool starts, all signal handlers are set to `SIG_DFL`. The action of `SIG_DFL` is as follows:

- Disable all signals
- Call the procedure `exit`

To specify your own signal handler procedure, call `signal` with your procedure as the `newHandler` parameter. When the signal is raised, your procedure will be called. Before your procedure is called, the `SIG_DFL` procedure is re-installed as the handler for that signal. Therefore, if you want to continue handling the signal, your procedure must re-install itself with another call to `signal` at the end of your signal handler.

▲ **Warning**   Because `SIG_DFL` is re-installed as part of the signal-handling process, your tool could be interrupted by a second signal that would then call `SIG_DFL`. It is safest to disable further signals by calling `signal (SIG_IGN)` at the beginning of your handler. Then re-install the appropriate handler at the end. ▲

You can think of signals as operating at the interrupt level. Therefore, the safest signal handler would set a global flag, re-install itself, and return. Then in the main body of your code, you could check for the flag and do some appropriate actions.

If you want to terminate program execution because of a signal, do the following: In your signal handler, disable that signal (using `SIG_IGN`) and set a flag. In the main body of your code, you can do some cleanup procedures, and call `exit`.

If you install a signal handler for command period, you should return an exit code of –9 to the MPW Shell. (For information on returning exit codes, see "`Exit`.")

Signals cannot be raised while executing in ROM or in the MPW Shell. If a signal event occurs while executing outside the tool, the signal state is set and the signal handler is executed as soon as program control returns to the tool. Because a signal can interrupt the tool at any point, there is no protection against heap corruption if a signal handler executes calls that modify the state of the heap. Because most buffered I/O potentially modifies the heap, writing to standard out or standard error is *not recommended* in signal handlers.

If you must perform I/O or other operations as a result of a signal, set a flag and check the flag during your own processing loop.

Chapter 13 **Creating a Commando Interface for Tools**

YOU CAN CREATE A COMMANDO DIALOG INTERFACE FOR YOUR OWN MPW TOOLS AND SCRIPTS. This chapter is a guide to creating the resources Commando requires to operate dialogs. The basic use of Commando's dialogs to operate MPW tools is described in Chapter 4. ■

Contents

About Commando

Commando makes it easier to use the MPW tools and scripts, both interactively and for composing scripts. A **dialog** is the programmed interaction between a user and a tool. A **dialog box** is the graphical vehicle used to display the various controls available for a tool or script. A dialog may employ several nested dialog boxes.

You implement the dialog interface for MPW tools by using Commando. This program looks in the resource fork of a tool or script for a resource of the type 'cmdo', that is, any dialogs to be used by the tool. Commando then loads the resource, builds a dialog list, handles events, and passes the command line back to the Shell for execution.

Invoking Commando

You can invoke a Commando dialog from the Worksheet in three ways:

■ **Option-Enter:** Type the command name and then press Option-Enter. This is the easiest method for routine interactive use.

■ **Type Commando:** Type the word Commando in front of the command line and press Enter. The Commando dialog outputs the command line without executing it. You can also use this expression in a script. For example, if you don't want the resulting command line to be immediately executed, you can type

commando *toolname*

In this case, Commando will not find a command if the command has been aliased to a different name. The tool's frontmost Commando dialog box is displayed. Clicking the Do It button writes the command line to standard output (that is, the window in which you typed the command) instead of executing it immediately. This second method of using dialog boxes is useful for building command lines that are to be cut and pasted into scripts. (The Do It button and other Commando controls are described later in this chapter.)

■ **Ellipsis:** Type the command name followed by an ellipsis (...) and press Enter. You can also use this expression in a script.

The ellipsis may appear anywhere in a command line (except within quotation marks or after ∂) and it is considered a word-break character. The ellipsis invokes the Commando user interface after the Shell has carried out all alias and variable substitutions. The entire command line is passed to Commando; the output of Commando is then executed by the Shell.

◆ *Note:* To get the ellipsis character, hold down the Option key while simultaneously typing the semicolon (;) character. Although three periods closely resemble an ellipsis character, Commando won't be fooled; you *must* use Option-semicolon to get the true ellipsis character that invokes Commando.

Three Shell variables are used by Commando:

■ **Aliases:** This variable lists all defined aliases, with each name separated by a comma. The list contains only the names, not the definitions. Commando uses {Aliases} with the built-in command Alias. Without this variable Commando would have no way of knowing the names of the existing aliases. The variable {Aliases} is exported by the Startup script.

■ **Commando:** This variable tells the Shell which command to execute when the ellipsis character is present in a command line. To use the Commando tool, set the variable {Commando} to "Commando." You can use this variable to send the output of other tools to the Shell for execution. If the variable does not exist, then the ellipsis is removed from the command line and normal execution proceeds.

■ **Windows:** This variable lists the current windows, with each name separated by a comma. Commando uses this list to redirect input or output to or from existing windows. Without this variable Commando would have no way of knowing the names of the current windows. The variable {Windows} is exported by the Startup script.

Throughout this chapter, each type of Commando control is illustrated with an excerpt from Cmdo.r, found in the RIncludes folder.

Creating Commando dialogs

Here is a procedure for creating your own Commando dialogs:

1. Create a Commando resource for your tool or script by starting with one of the example Commando resources, such as Count.r in CExamples or ResEqual.r in PExamples. If an existing tool has a Commando control that you want to use, derez (that is, decompile by using the resource decompiler DeRez) the cmdo resource, then cut it and paste it into your new Commando resource file.

   For example, to examine the Pascal compiler's cmdo resource:

   ```
   DeRez {MPW}Tools:Pascal -only cmdo cmdo.r
   ```

2. Add the Commando resource to your tool or script. For example,

   ```
   Rez   AddMenu.r   -o   "{MPW}MPW Shell"   -a
   Rez   Rez.r   -o   {MPW}Tools:Rez   -a
   ```

3. Now display the Commando dialogs for your tool or script. Adjust the coordinates of windows and move or resize the controls by using Commando's editor (see the next section). Edit the Help messages and then derez the `cmdo` resource.

Editing Commando dialogs

In MPW 3.0, Commando offers a built-in editor that lets you edit text labels and help messages and graphically move and size the controls within a Commando dialog box. This feature makes designing, redesigning, and fine-tuning Commando dialogs much easier. Although Commando can move and size controls, controls cannot be created, duplicated, or deleted. This means that you still have to manually create the Commando resource, but you don't have to be too concerned about the coordinates and sizes of the controls. Once you've created the Commando resource, you can simply bring up the Commando dialog in edit mode, arrange all the controls to your liking, and then use DeRez to decompile the `cmdo` resource.

Enabling Commando's Editor

To enable Commando's built-in editor, hold down the command key immediately after launching Commando until the Watch cursor appears. Alternatively, you can write **-modify** in the command line, like this

```
Commando Rez -modify
```

or

```
Rez… -modify
```

Editing controls

After you have launched Commando with the built-in editor enabled, it can run in either of two different modes:

■ Normal mode, in which Commando works as usual

■ Edit mode, in which controls can be dragged and sized. Hold down the Option key to put Commando in edit mode. You must also hold down the Option key to select, drag, or resize a control.

Selecting controls

To select a control, simply press the Option key and click the control. To select multiple controls, press the Option and Shift keys together and click each control to be selected. You can also click and drag a marquee around a group of controls, as you would in a paint program. To unselect a control, click it with the Shift (and Option) key down.

Basically, selecting controls works exactly like selecting icons in the Finder, except that you must hold down the Option key in Commando. Also, the Commando editor will not allow you to select controls outside the user control area. For that reason, the coordinates you give when manually creating the Commando resource should fall within the user area.

Moving controls

Moving controls works as you would expect: hold down the option key as you click and drag a control or a selected group of controls. The Commando editor will not allow controls to be dragged outside the user control area. Controls cannot be dragged closer than two pixels from the boundary.

You can move selected controls one pixel at a time by holding down the option key and pressing the appropriate arrow key.

You can align the top-left corner of the control to a four-pixel grid by holding down the command key while dragging. If you drag a selected group of controls with the command key held down, then the top-left corners of *each* of the selected controls will be aligned to the grid.

Sizing controls

You size controls by dragging the small gray rectangle in the control's lower-right corner while holding down the Option key.

Hold down the command key while sizing a control to size the control's height to the recommended Commando height (the sections that follow recommend a height for each control that works and looks best). Also, the right edge will align to a four-pixel grid. To size the control, simply click the selected control's grow handle with Option and Command keys held down. List and MultiRegularEntry controls will be sized to the nearest whole line.

Some controls, such as Redirection controls, cannot be resized and have no grow handles.

Editing labels

To edit a text title label, simply select it in the same way you would select a control. You can change the text in the same way you change the text for an icon in the Finder. Once the title is selected, don't hold down the Option key to change the text. Text title labels are the only labels that can be edited.

Editing Help messages

Whenever you select a control, the control's Help message is locked in the Help window. By clicking inside the Help window you can edit the Help message in the same way you edit a regular text edit field except that you don't hold down the Option key. The Help message stays locked until another control is selected (and then the new control's Help message is locked) or until all the controls are unselected.

Changing the size of a Commando dialog box

Once you have enabled Commando's built-in editor, you can resize any Commando dialog box by holding down the Option key while clicking and dragging by the dialog box's lower-right corner (where you would ordinarily find a Grow box in a standard Macintosh window). You can also resize nested dialog boxes. However, dialog boxes cannot be enlarged beyond the size of the original Macintosh screen.

◆ Hints and kinks

Lines and boxes surrounding other controls must be declared later in the Commando resource than the controls they surround. You may encounter situations in which you have to move a control out of the way in order to select a control underneath.

Controls sized or moved in nested dialogs do not go back to their original size or position when you click the nested Cancel button. ◆

With the Commando editor enabled, any text entered in a Regular Entry or MultiRegular Entry field is saved as the default text. The text will appear the next time Commando is invoked. See the sections on Regular Entry and MultiRegular Entry controls later in this chapter.

Saving the modified Commando dialog

Once you've modified a Commando dialog and clicked the main cancel or Do-It button, Commando prompts you with a Save dialog. The Save dialog has three options:

1. Save the resource.
2. Don't save the resource.
3. Cancel the Save dialog and go back to Commando for more editing.

When Commando saves the resource, it simply replaces the original resource, wherever it came from. The next time you run Commando on the changed resource, the control positions and sizes will be where you last left them. You can then use DeRez to decompile the cmdo resource to get the actual coordinates or to generate the .r file that will be used in a build.

Strings and Shell variables

You can dynamically change strings in Commando dialogs by having those strings come from Shell variables. To make strings come from Shell variables, define the string like this:

`"{shell variable}"`

The string must begin with a

`'{'`

and end with a

`'}'`
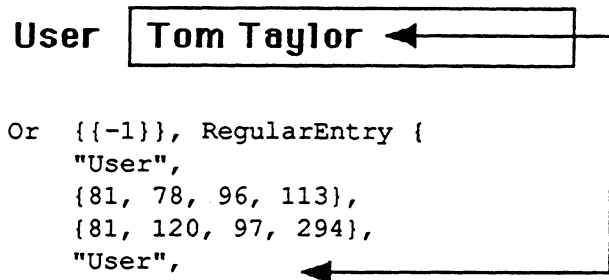
No leading or trailing spaces are allowed. The Shell variable must be an exported variable. If the variable is undefined at the time the Commando dialog is invoked, the variable name with braces will be displayed.

Any string in the Commando resource, including option strings, help strings, titles, and so on, can be a Shell variable. However, strings cannot be embedded within strings; they are stand-alone only.

When Commando is invoked with its built-in editor, Shell variable strings are not expanded to the Shell variable values. This is done so that the strings can be edited and then saved as Shell variables rather than the values of Shell variables.

Incidentally, this feature has been used in some of Projector's Commando dialogs in order to display the current user, as shown in Figure 13-1.

■ **Figure 13-1** Example use of the {User} variable

```
User  | Tom Taylor ◄──────────────────┐
                                       │
Or  {{-1}}, RegularEntry {            │
    "User",                           │
    {81, 78, 96, 113},                │
    {81, 120, 97, 294},               │
    "User",         ◄─────────────────┘
    ignoreCase,
    "-u",
    "Enter the name of the current user.  If
    "no name is entered, the name in {User} is used."
{,
```

Resource description

Cmdo.r, the resource description file for Commando, is located in {RIncludes}cmdo.r.

Resource ID and name

Any resource ID may be used for tools or scripts. Commando uses the first 'cmdo' resource it finds in the command's resource fork.

Commando draws an outlined button, the Do It button, in the lower-right corner of every dialog box. The Do It button is labeled with the name of the tool or script. (Normally Commando uses the name of the tool or script passed from the Shell.) Commando will capitalize the first character and force the rest of the characters to lowercase. For example, "StackSNiffER" becomes "Stacksniffer."

Some people may prefer a different capitalization scheme. If you specify the 'cmdo' resource of a tool or script with a resource name, Commando will use that name "as is" as the label for the outlined button. This feature should be used sparingly; if you rename a tool, the previous name in the resource will still be displayed in the Do It button.

Size of the dialog box

The width of Commando dialog boxes is fixed at 480 pixels. You are free to set the height to accommodate the controls in your tool's dialog box. The number specifying the height shouldn't exceed 295 to be compatible with the smaller Macintosh screens. Specifying this height in the `'cmdo'` resource will result in the screen elements shown in Figure 13-2. See Table 13-1 for other recommended dimensions.

Please refer to the array declared under " `resource 'cmdo'` " in the sample resource description file at the end of this chapter. The area labeled "Options" in Figure 13-2 is the area reserved for your controls and options.

■ **Figure 13-2**   Basic template for a Commando dialog box
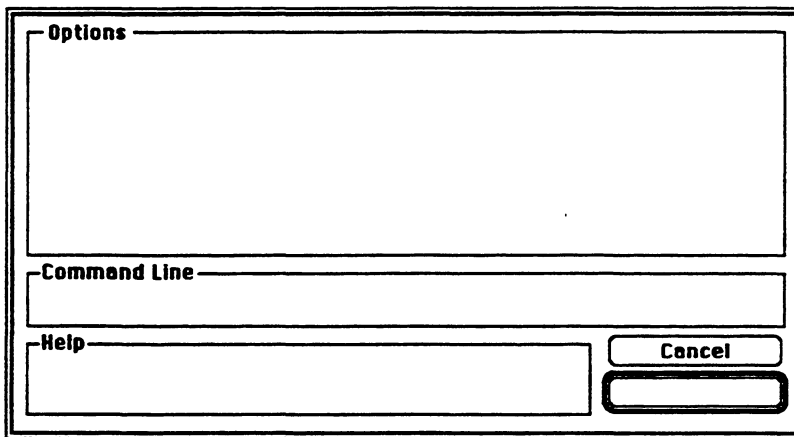
The dimensions given below are not policy but recommendations. The sizes of the text-edit fields are important if you want to avoid text that shifts up and down slightly when it is selected.

■ **Table 13-1**   Summary of recommended sizes for Commando screen elements

Screen element	Recommended size
Radio buttons	16 pixels high
Check boxes	16 pixels high
Pop-up menus	19 pixels high
Pop-up menu titles	16 pixels high. Top of title starts 1 pixel below the top of the pop-up menu (that is, top of title = top of pop-up menu + 1 pixel).
Regular entries	16 pixels high
Multi-regular entries	16 pixels per line
Editable pop-up menus	22 pixels high
Editable pop-up titles	16 pixels high. Top of title starts 3 pixels below the top of the editable pop-up menu (that is, top of title = top of editable pop-up + 3 pixels).
Icons	32 pixels high; 32 pixels wide
Pictures	Same relative bounds as the rectangle stored in the 'PICT' resource

Tool description

At the bottom of the Commando dialog box is a three-line Help box. The text in this box should be a brief, concise description of the tool, stating what it does. The Help box is not scrollable, so you need to limit your text to the confine of the box. See the array declared under "resource 'cmdo' " in the sample resource description file at the end of this chapter.

Regular entry control

The regular entry control is the most generic control available. The control behaves exactly like the text-edit fields in conventional Macintosh dialog boxes. In addition to strings and numbers, the regular entry control can be used for special options that have no specified standard control.

Here is the case declaration for regular entry controls:

```
case RegularEntry:
   key byte = RegularEntryID;
   cstring;                            /* title */
   align word;
   rect;                               /* bounds of title */
   rect;                               /* bounds of input box */
   cstring;                            /* default value */
   byte ignoreCase  keepCase;          /* the default value is never
                                       displayed in the Command
                                       window. If what the user
                                       types in the textedit window
                                       matches the default value,
                                       then that value isn't
                                       displayed. This flag tells
                                       Commando whether to ignore
                                       case when comparing the
                                       contents of the text edit
                                       window with the default value
                                       */
   cstring;                            /* option returned.*/
   cstring;                            /* help text for entry */
```

Multiregular entry

Multiregular entry controls are similar to regular entry controls, except that multiregular entry controls accept values that can be entered more than once. For example, most compilers accept some type of -define option that can be specified more than once.

Here is the case declaration for MultiRegular Entry. Note that the cstring for default values is the only control that passes its default values to the command line. This is an exception to the rule.

```
case MultiRegularEntry:              /* scrollable lists of an option */
    key byte = MultiRegularEntryID;
    cstring;                         /* title */
    align word;
    rect;                            /* bounds of title */
    rect;                            /* bounds of input list */
    byte = $$CountOf(DefEntryList);
    array DefEntryList  {
       cstring;                      /* default values */
    };
    cstring;                         /* option returned. Each value will
                                     be preceded with this option.*/
    cstring;                         /* help text for entry */
```

Figure 13-3 shows the Defines window in the Rez dialog box with two defines entered. Here is the resource control for this function:

```
NotDependent {}, MultiRegularEntry   {
    "Defines:",
    {20, 35, 35, 125},
    {40, 30, 120, 225},
    {},
    "-d",
    "Type in multiple #defines here   (such as LANGUAGE=French)"
},
```

The empty braces after the Defines window coordinates indicates that there are no default strings.

Defines:
```
debug=true
language=french
```

Command Line:
```
rez -d debug=true -d language=french
```

Check boxes

The check box control is likely to be the most often used because it corresponds to the on/off type options typical of MPW tools. Here is the case declaration for CheckOption:

```
case CheckOption:
    key byte = CheckOptionID;
    byte NotSet, Set;      /* whether button is set or not */
    rect;                  /* bounds */
    cstring;               /* title */
    cstring;               /* option returned */
    cstring;               /* help text for button */
```

The byte NotSet or Set is used to set the button's default state. The option is returned only when the button is not its default state.

This resource code produces the check boxes in Figure 13-4:

```
notDependent { }, CheckOption {
    NotSet, {20, 10, 36, 235}, "Show macro expansions",
        "-print GEN",
    "Expand macros in the listing file." },
notDependent { }, CheckOption {
    Set, {35, 10, 51, 235}, "Allow automatic page ejects",
        "-print NOPAGE",
    "Controls whether the Assembler sends automatic page ejects
        to the listing file" },
```

```
notDependent { }, CheckOption {
        Set, {50, 10, 66, 235}, "Show warning mesages",
                "-print NOWARN",
        "Controls both the display and count of warning messages." },
notDependent { }, CheckOption {
        Set, {65, 10, 81, 235}, "Show macro call statements",
                "-print NOMCALL",
        "Controls the listing of macro call statements." },
notDependent { }, CheckOption {
        Set, {80, 10, 96, 235}, "Show generated object code",
                "-print NOOBJ",
        "List generated object code or data for each listed line." },
notDependent { }, CheckOption {
        NotSet, {95, 10, 111, 235}, "Show up to 255 bytes of data",
                "-print DATA",
        "Controls whether object data is shown in full
                (up to 18 lines) or limited to one line." },
notDependent { }, CheckOption {
        Set, {110, 10, 126, 235}, "Show macro directive lines",
                "-print NOMDIR",
        "Controls whether macro directives (including conditional
                and set directives) are shown in the listing." },
notDependent { }, CheckOption {
        Set, {125, 10, 141, 235}, "Show header lines",
                "-print NOHDR",
        "Controls whether header lines are printed in the listing." },
notDependent { }, CheckOption {
        Set, {140, 10, 156, 235}, "Show generated literals",
                "-print NOLITS",
        "Controls listing of literals produce by PEA and LEA machine
                instructions." },
notDependent { }, CheckOption {
        NotSet, {155, 10, 171, 235}, "Show assembly status",
                "-print STAT",
        "Controls display of assembly status in the listing." },
```

Figure 13-4 shows a set of check boxes in their default state and again after the two top buttons have been clicked.

- **Figure 13-4**   Setting the CheckOption default state

Default state of buttons

☐ Show macro expansions
☒ Allow automatic page ejects
☒ Show warning messages
☒ Show macro call statements
☒ Show generated object code
☐ Show up to 255 bytes of data
☒ Show macro directive lines
☒ Show header lines
☒ Show generated literals
☐ Show assembly status

┌Command Line:─────────
│asm

State after top two buttons clicked

☒ Show macro expansions
☐ Allow automatic page ejects
☒ Show warning messages
☒ Show macro call statements
☒ Show generated object code
☐ Show up to 255 bytes of data
☒ Show macro directive lines
☒ Show header lines
☒ Show generated literals
☐ Show assembly status

┌Command Line:─────────
│asm -print GEN -print NOPAGE

Radio buttons

The simplest set of radio buttons offers several mutually exclusive options. For example, the Print Option radio buttons in Figure 13-5 let you choose High or Standard or Draft. The Standard mode is the default.

- **Figure 13-5**   Radio buttons with default setting

○ High
◉ Standard
○ Draft

┌Command Line:──
│test

Here is the case declaration for radio buttons:

```
case RadioButtons:
    key byte = RadioButtonsID;
    byte = $$CountOf(radioArray);    /* # of buttons */
    wide array radioArray {
        rect;                         /* bounds */
        cstring;                      /* title */
        cstring;                      /* option returned */
        byte NotSet, Set;             /* whether button is set or not */
        cstring;                      /* help text for button */
        align word;
    };
```

To make the middle radio button the default, as shown in Figure 13-5, declare the middle Standard button set:

```
notDependent { }, RadioButtons {
    {
        {115, 300, 130, 400}, "High", "-q high", notset,
            "Print the selected files in the highest quality"
            "available from the printer.",
        {132, 300, 147, 400}, "Standard", "-q standard", set,
            "Print the selected files in the normal quality mode.",
        {149, 300, 164, 400}, "Draft", "-q draft", notset,
            "Print the selected files in the fastest way possible"
            "at the expense of quality."
```

No option is passed to the command line box because the middle button is explicitly declared the default. If a button other than the default is clicked, Commando passes the appropriate option to the command line, as shown in Figure 13-6.

- **Figure 13-6**   Clicking a button other than the default

◉ **High**
○ **Standard**
○ **Draft**

┌**Command Line:**──
│test -q high

Suppose that in the previous example you wanted the default radio button to display its option in the command line. You would simply change the order in which you declared the radio buttons, so that the middle button would be declared first. Be sure that all buttons are NotSet. The result is shown in Figure 13-7. Commando will set the first button that it encounters if no button is specified as set.

■ **Figure 13-7**   No button specified as set

○ **High**
◉ **Standard**
○ **Draft**

┌**Command Line:—**
│test -q standard

◆ *Note:* A radio button can be either dependent upon or parent to another control. For purposes of establishing dependency relations, a cluster of radio buttons is considered a single item in the resource listing. See "Control Dependencies" later in this chapter for more information.

Boxes, lines, and text titles

It is recommended that you group dialog controls or functions within boxes. Commando supplies the facilities to draw a box (case Box), to draw a box with a title embedded in the upper-left corner (case TextBox), and to create titles in any font (case TextTitle).

◆ *Note:* When you draw a box around a set of controls, always list the box declaration after listing the other controls. Otherwise the Dialog Manager might confuse which control is clicked.

Box and TextBox cannot depend on other controls, nor can other controls depend on them. Commando would not complain if you set up such a dependency, but the line or box would not respond to the state of the determining item. TextTitles, on the other hand, can be dependent on another control.

Box

Use the case Box to draw boxes around controls or to draw lines. To draw lines, make the rect 1 pixel wide or 1 pixel high. In other words, to draw a horizontal line, you might set the rect to {10, 10, 11, 100}. Here is the case declaration for the Box control:

```
case Box:              /* Can be used to draw lines too */
    key byte = BoxID;
    byte   black, gray;   /* color of object */
    rect;                 /* bounds of box or line */
```

TextBox

The case TextBox lets you draw a box with the title embedded in the line at the upper-left corner. This is a frequently used convention in the Commando dialogs. (See the sample dialog box template in Figure 13-2.) Here is the case TextBox declaration from the Command resource file:

```
case TextBox:          /* Draws a box with title in upper-left */
    key byte = TextBoxID;
    byte   black, gray;   /* color of object */
    rect;                 /* bounds of box or line */
    cstring;              /* title */
```

For example,

```
notDependent { }, TextBox {
    gray,
    {105, 295, 169, 405},
    "Print Quality"
} ,
```

This declaration gives you the results shown in Figure 13-8.

```
┌─Print Quality─┐
│ ○ High         │
│ ⦿ Standard     │
│ ○ Draft        │
└───────────────┘
```
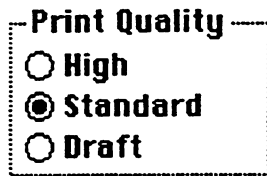
TextTitle

Use TextTitle to draw text in any font. Here is the case declaration:

```
case TextTitle:
    key byte = TextTitleID;
    byte  plain;              /* style of text */
    rect;                     /* bounds of title */
    int  systemFont;          /* font number to use */
    int  systemSize;          /* font size to use */
    cstring;                  /* the text to display */
```

For example, let's write "so cool" in a cool way:

```
notDependent { },  TextTitle  {
    bold + italic, {20,20,40,100},
    systemFont, 12, "So Cool..."
} ,
```

So Cool...

Pop-up menus

Pop-up menus are a convenient way to select an item from a list of related items. Commando manages the associated windows, aliases, fonts, and Shell variables. Here is the case declaration:

```
case PopUp:
    key byte = PopUpID;
    byte  Window, Alias, Font, Set;   /* popup type */
    rect;                             /* bounds of title */
    rect;                             /* bounds of popup line */
    cstring;                          /* title */
    cstring;                          /* option returned */
    cstring;                          /* help text for popup */
    byte  noDefault, hasDefault;      /* hasDefault if 1st item
                                         is "Default Value" */
```

The last field, "byte noDefault, hasDefault," tells Commando whether the pop-up menu has a default value or not. If the pop-up menu does not have a default value, the first value in the pop-up list is automatically selected and passed to the command line. If the pop-up menu does have a default value, then Commando adds a new item of the form "Default *Value*" to the front of the list. When this value (such as a font or file) is selected, no value is displayed in the window that generates the pop-up menu.

Here is an example of the resource code for a pop-up menu with a default value. See Figure 13-9 for the resulting window and pop-up menu.

```
notDependent { }, PopUp   {
    Font,
    {21, 20, 37, 60},
    {20, 60, 39, 160},
    "Font",
    "-f",
    "Popup help message",
    hasDefault
},
```

■ **Figure 13-9**   Pop-up menu with default value

Font ☐

```
✓ Default Font ▸
  Auant Garde
  Bookman
  Chicago
  Courier
  Geneva
  Helvetica
  Monaco
  N Helvetica Narrow
  New Century Schlbk
  Palatino
  Saigon
  Symbol
  Times
```

Here is the resource code for a pop-up menu with no default value. The results are shown in Figure 13-10.

```
notDependent { }, PopUp  {
    Font,
    {46, 20, 62, 60},
    {45, 60, 64, 160},
    "Font",
    "-f",
    "popup help message",
    noDefault
},
```

■ **Figure 13-10**   Pop-up menu without default value

Font [Auant Garde]

```
✓Auant Garde
  Bookman
  Chicago
  Courier
  Geneva
  Helvetica
  Monaco
  N Helvetica Narrow
  New Century Schlbk
  Palatino
  Saigon
  Symbol
  Times
```

Editable pop-up menus

Pop-up menus associated with a text-edit box can be edited. You can choose existing values from a list and still have the flexibility to enter completely new values.

```
case EditPopUp:
    key byte = EditPopUpID;
    /* For MenuItem, this EditPopUp must be dependent on
        another EditPopUp of the MenuTitle type so that
        the control recognizes which menu item to display.

        For FontSize, this EditPopUp must be dependent on
        a PopUp of the Font type so that the control
        recognizes which sizes of the font exist. */

byte    MenuTitle, MenuItem,    /* Type of editable pop-up */
        FontSize, Alias, Set;
rect;                           /* bounds of title */
rect;                           /* bounds of text edit area */
cstring;                        /* title */
cstring;                        /* option to return */
cstring;                        /* help text for text-edit */
cstring;                        /* help text for pop-up */
```

The example in Figure 13-11 shows how the Font Size editable pop-up menu is made dependent on the current font.

■ **Figure 13-11**   How Font Size dependency is handled


```
notDependent {}, PopUp {
Font,
{21, 20, 37, 60},
{20, 60, 39, 160},
"Font",
"-f",
"Popup help message.",
hasDefault
},
Or {{1}}, EditPopUp {
FontSize,
{48, 20, 64, 90},
{45, 90, 67, 140},
"Font Size",
"-size",
"Textedit help message",
"Popup help message"
},
```
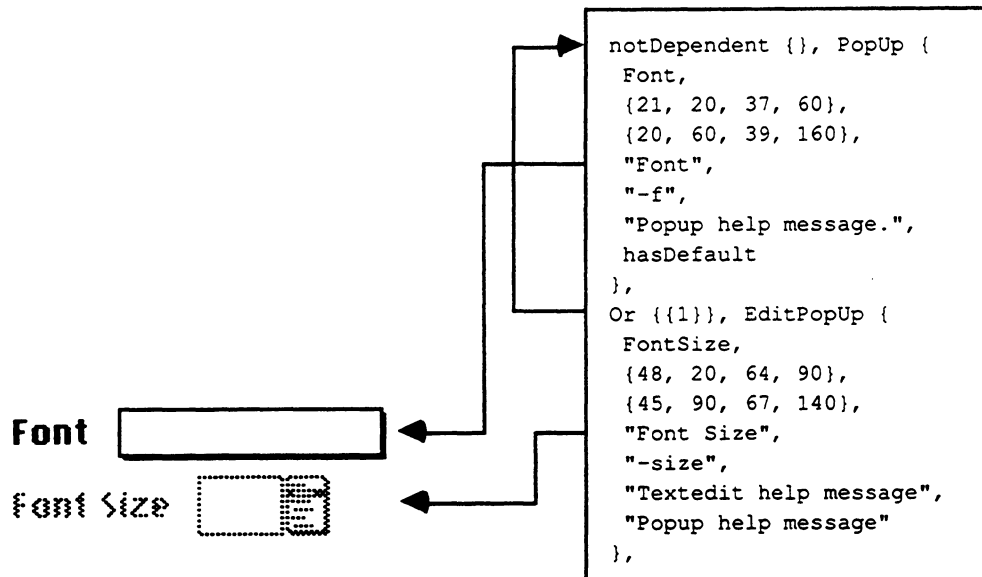
Font [   ]

Font Size [▦]

If a particular font is selected in the Font box, then the font sizes that actually exist are outlined. In the example in Figure 13-12 the Monaco font has been selected in the Font box. The 9-point item is outlined and has been selected with the mouse. Any font size can be typed in the Font Size box.

■ **Figure 13-12**   Font Size pop-up menu with font selected

Font |Monaco |

Font Size |9|

| ✓9 Point |
| 10 Point |
| 12 Point |
| 14 Point |
| 18 Point |
| 24 Point |
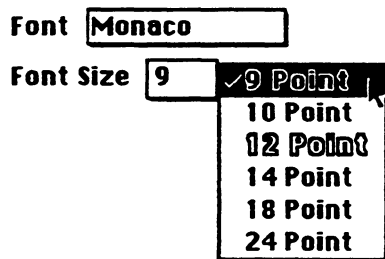
Figure 13-13 demonstrates how one editable pop-up menu can be dependent on another.

■ **Figure 13-13** One pop-up menu dependent on another

```
notDependent {}, EditPopUp {
  MenuTitle,
  {42, 98, 59, 142},
  {40, 129, 68, 259},
  "Menu",
  "-m",
  "Textedit help message",
  "Popup help message"
},
Or {{1}}, EditPopUp {
  MenuItem,
  {75, 100, 91, 136},
  {72, 129, 94, 259},
  "Item",
  "-i",
  "Textedit help message",
  "Popup help message"
},
```

Menu |

Item

File
Edit
Find
Mark
Window
Project
Tools
Cmds
Directory
Build
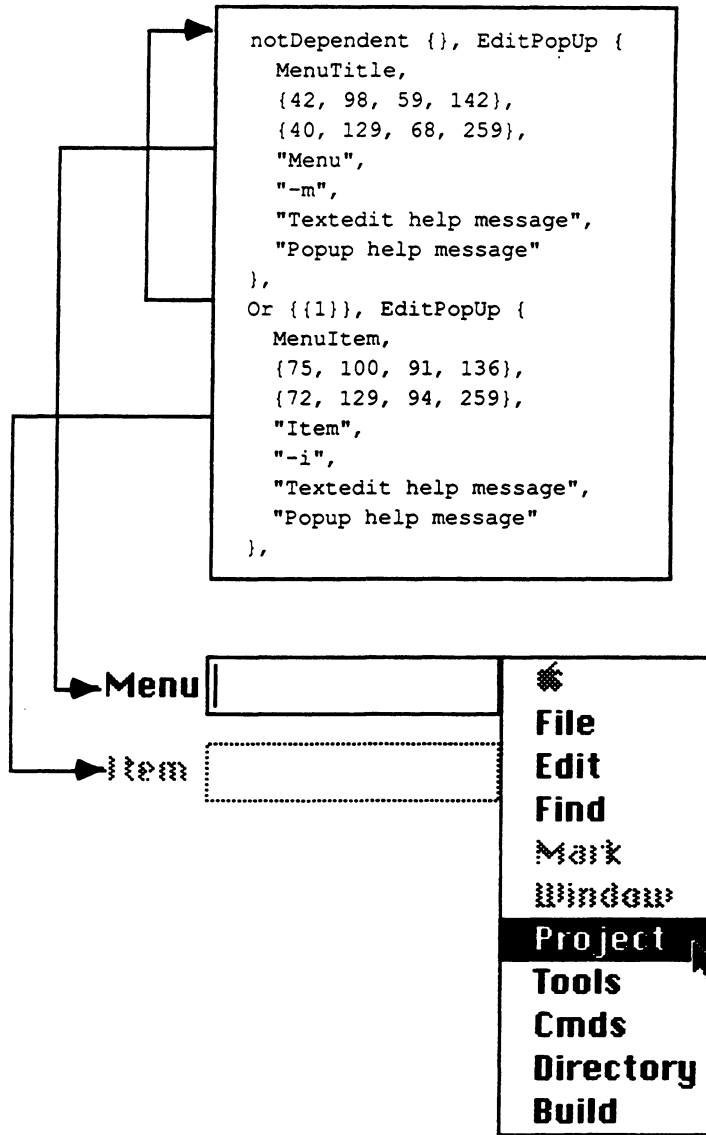
Because the MenuItem EditPopUp is dependent on the MenuTitle EditPopUp, the MenuItem control is dimmed until a menu is selected from the Menu pop-up or until a menu is typed in the Menu textedit field.

After "Project" is selected (Figure 13-13), the Item menu is enabled as shown in Figure 13-14.

■ **Figure 13-14**   Menu title and Item pop-up menus

Menu `Project`

Item

```
New Project
Check In
Check Out

AExamples
CExamples
PExamples
RIncludes

CheckOut {Active}
CheckIn {Active}
```
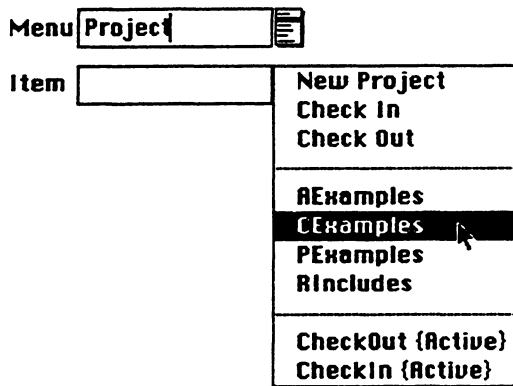
Lists

Use the case List to enable users to make multiple selections from a list of items. Four types of things can be listed:

■ volumes (Inserted disks will be recognized and added to the list.)

■ shell variables

■ windows

■ aliases

Here is the case declaration for List:

```
case List:                      /* Puts up list of items & allows
                                   multiple selections */
    key byte = ListID;
    byte  Volumes, ShellVars,   /* what to display in
          Windows, Aliases;        the list */
    cstring;                    /* option to return before each item */
    cstring;                    /* title */
    align word;
    rect;                       /* bounds of title */
    rect;                       /* bounds of list selection box */
    cstring;                    /* help text for selection box */
```

Here is the resource code for the two examples shown in Figure 13-15. The second example shows that the user has already selected a window.

```
notDependent { }, List {
    Volumes,
    "",
    "Volumes",
    {20,30,35,120},
    {37,30,101,200},
    "Help message"
},
notDependent { }, List {
    Windows,
    "-w",
    "Window List",
    {20,220,35,303},
    {37,220,101,400},
    "Help message"
},
```

■ **Figure 13-15**   List control
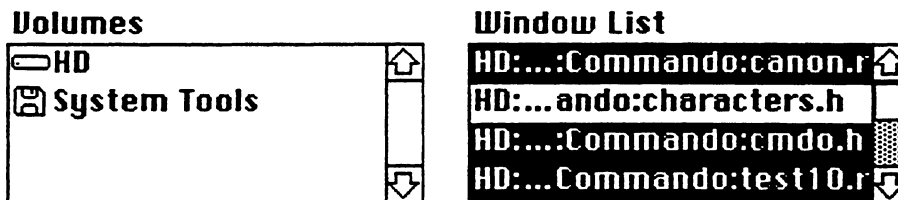
Three-state buttons

Three-state buttons were invented to handle the SetFile and SetPrivilege commands. Both of these commands deal with the setting or clearing of flags. These commands also have another implicit state: "Don't touch." Therefore, these buttons have three states: Set, Clear, and DontTouch.

```
case TriStateButtons:
    key byte = TriStateButtonsID;
    byte = $$CountOf (threeStateArray);     /* # of buttons */
    cstring;                                 /* option returned */
    wide array threeStateArray {
        align word;
        rect;                               /* bounds */
        cstring;                            /* title */
        cstring;                            /* for Clear state */
        cstring;                            /* for DontTouch state */
        cstring;                            /* for Set state */
        cstring;                            /* help text for button */
        };
```

Here is the resource code for the example shown in Figure 13-16.

```
notDependent { }, TriStateButtons {
    "-a",
    {
    {40, 25, 58, 135}, "Locked", "l", "", "L",
     "This button affects the file \"Locked\" attribute.",
    {58, 25, 76, 135}, "Invisible", "v", "", "V",
     "This button affects the file \"Invisible\" attribute.",
    {76, 25, 94, 135}, "Bundle", "b", "", "B",
     "This button affects the file \"Bundle\" attribute.",
    {94, 25, 112, 135}, "System", "s", "", "S",
     "This button affects the file \"System\" attribute.",
    {112, 25, 130, 135}, "Inited", "i", "", "I",
     "This button affects the file \"Inited\" attribute.",
    {130, 25, 148, 135}, "On Desktop", "d", "", "D",
     "This button affects the file \"On Desktop\" attribute."
    }
},
```

■ **Figure 13-16**   Three-state buttons

Icons and pictures

Use the case PictOrIcon to place icons, pictures, or both in the Commando windows. This item cannot be dependent on any other item, nor other items on it. Here is the case declaration for icons and pictures:

```
case PictOrIcon:
    key byte = PictOrIconID;
    byte Icon, Picture;        /* display a picture or icon */
    int;                       /* resource ID of icon */
    rect;                      /* display bounds */
```

The icon in Figure 13-17 is produced by an ˈICONˈ resource with an ID of 0, located in the system file.

■ **Figure 13-17**   Icon in a Commando window

Here is the resource code that generates the example shown in Figure 13-17:

```
notDependent, PictOrIcon {
    Icon, 0, {20, 20, 52, 52}
},
```

Control dependencies

Sometimes one control is dependent on the value of another control. For example, a font size control might be dependent on a preceding font selection control. In this case the font size control is termed the **dependent.** The preceding font selection control is called the **parent** because it enables or disables the dependent.

Commando numbers each item sequentially in the order of its appearance in the resource description file. The dependent/parent relationship in Commando is controlled by the sequential order of items entered into a Commando resource.

◆ *Note:* These numbers do not appear in the resource code; you must count them manually.

An item may be dependent only on other items within the same dialog box. In the case of nested dialog boxes, the items in the second and succeeding dialog boxes must be renumbered, starting from one.

Direct dependency

Usually dependency on another control means that the dependent control is disabled if the parent control is disabled (or has no value).

Figure 13-18 shows two states of a directly dependent control. In the first case, nothing has been entered in the Type field, so the dependent Creator field is disabled and appears dimmed in the dialog box. In the second case, the Creator field is enabled as soon as something is typed in the Type field.

Figure 13-18 also illustrates how the ignoreCase/keepCase flag works. Because the flag is keepCase and 'appl' is not equal to 'APPL' (the default value in this case), the option is displayed in the Command Line box.

■ **Figure 13-18**   Direct dependency

```
 |                      Type   ┌──────────┐
 |                             └──────────┘
 |                     Creator │ ???? │
 |_____     └┄┄┄┄┄┄┘
┌─Command Line:──────────────────────────┐
│test
│
```

```
 |                      Type   ┌──────┐
 |                             │ appl │
 |                             └──────┘
 |                    Creator  ┌──────┐
 |                             │ ???? │
 |_____     └──────┘
┌─Command Line:──────────────────────────┐
│test -t appl
│
```

Inverse dependency

A control can be inversely dependent on another control. In other words, if the parent is disabled, then the dependent is enabled. Or, if the parent is enabled, then the dependent is disabled.

It is also possible for two controls to be inversely dependent on each other. This means that both controls are enabled until one is selected; then the other is disabled. For example, there are two types of dependencies illustrated in Figure 13-19. The user can select either the top check box or the bottom one, but not both; that is, the user is allowed to append resources to a resource file *or* to make the resource map read-only. The middle box is enabled only when the top box is checked, because it makes sense to replace protected resources only when appending to a source file.

■ **Figure 13-19** Inverse dependencies

☐ **Append resources to resource file**
☐ **OK to replace protected resources**
☐ **Make resource file read-only**

☒ **Append resources to resource file**
☐ **OK to replace protected resources**
☐ Make resource file read-only

☐ Append resources to resource file
☐ OK to replace protected resources
☒ **Make resource file read-only**

Here is the resource description of the three check boxes shown in Figure 13-19. To make a control inversely dependent on another control, make the value of the parent negative.

```
Or { {-3} }, CheckOption {
    NotSet,
    {20, 10, 40, 350},
    "Append resources to resource file",
    "-a",
    "some help text..."
},
Or { {1} }, CheckOption {
    NotSet,
    {40, 10, 60, 350},
    "OK to replace protected resources",
    "-ov",
    "some help text..."
},
Or { {-1} }, CheckOption {
    NotSet,
    {60, 10, 80, 350},
    "Make resources file read-only",
    "-ro",
    "some help text..."
},
```

The second CheckOption (the dependent) is enabled only if the first (the parent) is enabled. The third CheckOption is enabled only if the first is disabled.

Dependency on the Do It button

To make the Do It button dependent on something, you must use the special Do It
Button item in the Commando resource type definition. This item can be specified only
once per resource and can be specified only in the first dialog. In the example shown in
Figure 13-20 the Do It button is dependent on the check box.

■ **Figure 13-20** Dependency on the Do It button

Here is the resource code for the above example:

```
NotDependent { }, CheckOption {
        NotSet,
        {20, 20, 40, 200},
        "Check me baby",
        "-c",
        "Help us to help you.",
},
Or { {1} }, DoItButton {
}
```

Multiple dependencies

A Commando item can be dependent on one or more other items. For example, a control
might be enabled only when two other controls are enabled. Such situations are
considered multiple dependencies.

Multiple dependencies may be of two types: OR and AND. In an OR dependency, a
dependent control is enabled if any of its parents is enabled. In an AND dependency, the
dependent control is enabled only if all its parents are enabled. It is possible to mix ANDs
and ORs. For example, include an item within an AND or OR list that is dependent on a
dummy control (case Dummy)—and make the dummy control dependent on another list
of controls. An example appears in the next section.

Dependencies on radio buttons

Commando considers a cluster of radio buttons to be one item. Remember that
Commando numbers each item sequentially in the order of its appearance in the resource
description file. When an item is dependent on a specific radio button within a cluster of
radio buttons, the number of the individual button is placed in the upper four bits of the
item number that describes the entire cluster of radio buttons. For example, consider a
radio button cluster that is item #5 and contains six radio buttons. To have a dependency
on button #3 you would write, in Rez syntax,

```
(3<<12) + 5
```

Figure 13-21 shows three ways in which the check box at the bottom of the dialog box is
dependent on the upper check box and radio buttons.

- **Figure 13-21** Dependencies on radio buttons

Here is the resource description code describing the operation of the dialog box in Figure 13-21:

```
notDependent { }, CheckOption {
       NotSet, {15, 15, 31, 100}, "Check Me", "-root", ""
},
And { {1, 3} }, CheckOption {
       NotSet, {65, 15, 81, 450}, "Depends on box above and"
       "buttons 1 & 3",  "-above1", ""
},
Or { { (1 << 12) + 4, (3<< 12) + 4 } }, Dummy {
},
notDependent { }, RadioButtons { {
       {15, 150, 31, 450}, "button 1", "-b1", NotSet, "no help",
       {30, 150, 46, 450}, "button 2", "-b2", NotSet, "no help",
       {45, 150, 61, 450}, "button 3", "-b3", NotSet, "no help",
} },
```

In Figure 13-21 the first CheckOption is Item #1 in the resource description file and the next CheckOption is Item #2 in the same file. Item #3 is a dummy item used to perform the complex dependency. Item #4 is the entire cluster of three radio buttons. Item #2 (the bottom check box in the sample dialog) is dependent on Item #1 (the top check box) AND radio button #1 OR radio button #3.

Nested dialog boxes

Complex tools may require more than one dialog box in order to display all the options. When there are several nested dialog boxes, all of them are called from buttons in the first dialog box. It's best to avoid calling nested dialog boxes from other nested dialog boxes.

Figure 13-22 shows how dialog box #2 can be called from dialog box #1.

■ **Figure 13-22** Setting up nested dialog boxes

```
wide array itemArray {
 int  notDependent = 0;  /* item dependent upon */
 switch {
  case NestedDialog:
   key byte = NestedDialogID;
   byte;      /* the number of the dialog
                 to call.  It must be greater
                 than the current dialog */
   rect;      /* bounds of button */
   cstring;   /* button's title */
   cstring;   /* help text for button */
```

Dialog #1 calls dialog #2.

```
resource 'CMDO' (128) {
 {
  270,
  "",
  {
   notDependent {}, NestedDialog {
    2,
    {135, 357, 155, 468},
    "Nested Dialog_",
    "This is the help message displayed when the nested dialog button is clicked."
   },
  },
```

Dialog #1

```
  270,
  ""
  {
   }
  }
};
```

Dialog #2

All items in a nested dialog box have an implied dependency on the nested dialog button. When a nested dialog button is disabled (dimmed), all the controls in that nested dialog act as if they were disabled.

Figure 13-23 shows the recommended placement of nested dialog call buttons.

■ **Figure 13-23** Placement of nested dialog buttons

User Interface recommends that nested dialog buttons begin at the lower-right and go up.

While the mouse is held down over any control, help info is displayed here.

Clicking the Cancel button in a nested dialog box reverts all its controls to their state before the nested dialog box was opened, thus returning the user to dialog box #1. Clicking the Do-It button (typically labeled "Continue") saves the current state of all controls in the nested dialog box, and returns the user to the first dialog box.

Redirection

Redirection is the easiest control to add to a Commando resource description file. Simply specify the type of redirection desired and the point location of the upper-left corner. Commando takes care of the rest. Here is the case declaration for redirection:

```
case Redirection
     key byte = RedirectionID;
     byte StandardOutput,      /* the type of redirection */
          DiagnosticOutput,
          StandardInput;
     point;                    /* upper-left point of the entire contraption *
```

Figure 13-23 shows the resource code for Redirection along with its results.

■ **Figure 13-24** How to obtain input and output redirection

```
notDependent {}, Redirection {
 StandardInput,
 {15, 27}
},
notDependent {}, Redirection {
 StandardOutput,
 {15, 252}
},
```

Input

Output

Click here and
get this pop-up

✓ *No Input Redirection*	✓ *No Output Redirection*
Existing File...	New File...
Window...	Existing File...
Current Selection in Window...	Window...
Current Selection in Target Window	Current Selection in Window...
Standard Input	Current Selection in Target Window
Null Device	Standard Output
Console Device	Standard Diagnostic
	Null Device
	Console Device

Files and directories

There are four types of Commando dialogs, offering four different ways to make files and directories available:

- as individual items for both input and ouput
- as multiple files for input only
- as multiple files and directories for input only
- as multiple new files for output

Input only means that a standard file dialog box is displayed when the command requires a file or directory on which to act. *Input or Output* allows the user to write over an existing output file without going through the standard file dialog.

Individual files and directories

The Files control enables users to select a single file or directory that can be used for input or output. This control supports seven combinations of files, as illustrated in Figure 13-25.

The files control is used to select a single file
or directory that can be used for input or output
This control supports seven combinations of files:

```
case Files:
key byte - FilesID;
byte InputFile,                 /* These types require the */
     InputFileOrDir,            /* 'Additional' case below. */
     InputOrOutputFile,
     InputOrOutputOrDir,

     OutputFile,                /* These types require the */
     OutputFileOrDir,           /* 'NoMore' case below.     */
     DirOnly;
switch {
case OptionalFile:
   /* Using this case makes a popup with two or three items.
      This first item is used to select a default file or to
      select no file.  The second (and third) item(s) are used
      to select input or output files. */
   key int - 0;
   rect;                        /* bounds of title */
   rect;                        /* bounds of display box */
   cstring;                     /* title */
   cstring;                     /* default file */
   cstring;                     /* option to return before file */
   cstring;                     /* If this item is dependent upon
                                   another Files item, an extension
                                   can be specified here to be added
                                   to the dependents file. */
   cstring;                     /* help text for popup */
   byte dim, dontDim;           /* Normally, dependent items are
                                   dimmed if the parent is disabled,
                                   if this field is "dontDim", then
                                   this item won't be dimmed */
   /* These next three strings are the strings displayed in the popup.  Most
      of the file types have only two strings but InputOrOutputFile and
      InputOrOutputOrDir require three strings.  If a string is set to "",
      Commando will use a built-in default. */
   cstring;                     /* popup item #1 */
   cstring;                     /* popup item #2 */
   cstring;                     /* popup item #3 */

case RequiredFile:
   /* Using this case makes a button that goes directly to standard
      file.  Use this case when a file is required and the user doesn't
      have the choice of a default file or no file. */
   key int - 1;
   rect;                        /* bounds of button */
   cstring;                     /* title of button */
   cstring;                     /* option to return before file */
   cstring;                     /* help text for button */
};

/* Some file types take additional information.  See the comment next to the
   file types to find out which case to choose here. */
switch {
case Additional:
   key byte - 0;
   cstring;                     /* For InputOrOutputFile, an option
                                   can be specified when a new
                                   (or output) file is chosen. */
   cstring FilterTypes - ":";/* preferred file extension (i.e. ".c")
                                   If null, no radio buttons will be
                                   displayed.  If FilterTypes is used,
                                   the radio buttons will toggle
                                   between showing files only with
                                   the types below and all files */
   cstring;                     /* title of only files button */
   cstring;                     /* title of all files button */
   byte - $$CountOf(TypesArray);/* up to 4 types may be specified */
   align word;
   array TypesArray {
   literal longint text - 'TEXT',  /* desired input file type, specify */
                     obj  - 'OBJ ',  /* no type for all types */
                     appl - 'APPL',
                     da   - 'DFIL',
                     tool - 'MPST';
   };

case NoMore:
   key byte - 1;
};
```

This case generates a popup.

This case generates a button.

Each group of files uses its own case

Here is the resource code for the "individual files and directories" controls that appear in Figure 13-25.

```
notDependent { } , Files {
        InputFile,
        OptionalFile {
                {20,20,40,130},
                {20,100,40,300},
                "C Input Files",
                "", "", "",
                "Help message here.",
                dim,
                "Read Standard Input",
                "Select a file to compile...",
                "",
        },
        Additional {
                "",
                ".c",
                "Only files that end in .c",
                "All text files",
                {text}
        },
},
Or {{1}}, Files {
        OutputFile,
        OptionalFile {
                {50,20,70,100},
                {50,100,70,300},
                "Object File",
                "c.o", "-o", ".o",
                "Help message here.",
                dontDim,
                "Send object code to c.o",
                "Select an object file...",
                "",
        },
        NoMore {},
},
```

Figure 13-26 shows the control resulting from the resource code above. The control is shown first in its default state, then as it appears after the user selects an input file, and finally as it appears after Commando produces the object file associated with the input file selected by the user.

■ **Figure 13-26** Examples of "individual files and directories" controls

Default state

```
C Input File [                              ]

Object File [c.o                            ]
```

Choose an input file

```
C Input File [✓Read Standard Input         ]
             [ Select a file to compile... ]
Object File  [c.o            ▶             ]
```

Object file dependent on input

```
C Input File [hd:...:Commando:checkBox.c   ]

Object File  [hd:...Commando:checkBox.c.o  ]
```

Multiple files and directories for input and output

Use the case MultiFiles (shown here) to enable users to select multiple files and directories for input and output. Note the four cases representing subtypes within the case MultiFiles:

■ case MultiInputFiles

■ case MultiDirs

■ case MultiInputFilesAndDirs

■ case MultiOutputFiles

Here is the MultiFiles case:

```
case MultiFiles:
    key byte = MultiFilesID;
    cstring;                    /* button title */
    cstring;                    /* help text for button */
    align word;
    rect;                       /* bounds of button */
    cstring;                    /* message like "Source files
                                    to compile:" */
...Cstring;                     /*  option returned before each filename
                                    Can be Null */
    switch {
case MultiInputFiles:
    key byte = 0;
    byte = $$CountOf (MultiTypesArray);    /* up to 4 types may be
                                               specified */
    align word;
    array MultiTypesArray {
       literal longinit  text = 'TEXT',    /* desired input file
                                               type, specify no type */
                         obj = 'OBJ ',      /* for all types */
                         appl = 'APPL',
                         da = 'DFIL',
                         tool = 'MPST';
};
cstring FilterTypes = ":";              /* preferred file extension
                                           (that is, ".c"). If null,
                                           no radio buttons will be
                                           displayed. If FilterTypes
                                           is used, the radio buttons
                                           will toggle between show-
                                           ing files with only the types
                                           below, and all files. */
cstring;                                /* title of only files button */
cstring;                                /* title of all files button */
case MultiDirs:
    key byte = 1;
case MultiInputFilesAndDirs:
    key byte = 2;
case MultiOutputFiles:
    key byte = 3;
};
```

Figure 13-27 is a Files dialog box controlled by resource code using the MultiFiles case.
Here is the resource code:

```
notDependent {}, MultiFiles {
    "Description Files...",
    "Select resource input files to compile",
    {60, 330, 80, 468},
    "Resource Description Files:", "" ,
    MultiInputFiles {
        {text},
        ".r",
        "Files ending in .r",
        "All text files"
        },
},
```

The button "Resource Description Files..." is the Rez dialog box that displays
the large standard file dialog box shown in Figure 13-27. The last two titles refer to the two
radio buttons.

- **Figure 13-27** Example of multiple input files
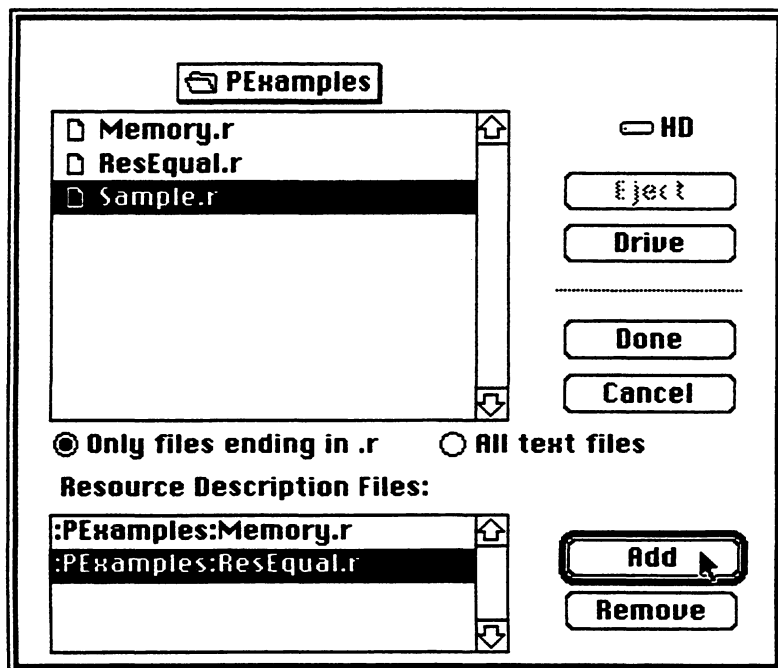
In the example in Figure 13-27 two resource files have just been added. When a file extension is specified, two radio buttons allow you to see only those files that have the specified extension or all files, regardless of their extension. In either case, only files that have a file type matching one of the those specified in the resource are displayed. Up to four file types may be displayed. If no file type is specified, all files are eligible for display.

If no file type or file extension is specified in the ' cmdo ' description, then no radio buttons are displayed, as shown in Figure 13-28:

```
notDependent {}, MultiFiles {
    "Files to delete…",
    "Select files to delete",
    {60, 330, 80, 468},
    "Files to delete:",
    "-d" ,
    MultiInputFiles {
        {},
        "",             /* no file extension specified */
        "",
        ""
    },
},
```

■ **Figure 13-28**   Example of multiple input files with no file extension specified
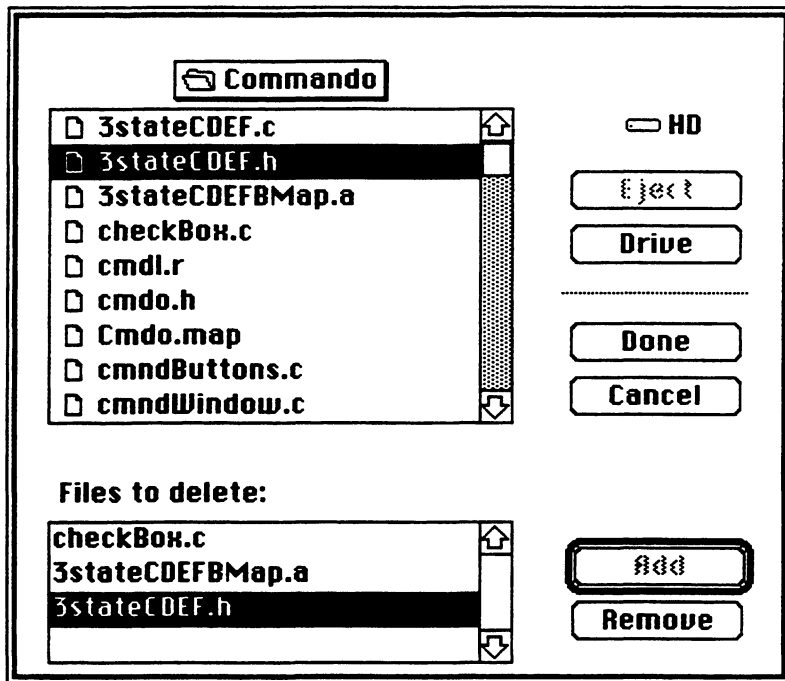
Sometimes the type of a file is more important than the file's extension. The Link tool, for example, identifies object files by the file type ('OBJ ') rather than by the file extension. By specifiying FilterTypes as the extension string, the radio buttons will toggle between showing files matching the specified types and showing all files, regardless of type. Here is an example of this behavior:

```
notDependent {}, MultiFiles {
    "Files to link…",
    "Select files to link",
    {60, 330, 80, 468},
    "Files to link:",
    "-l" ,
    MultiInputFiles {
        {'OBJ '},
        FilterTypes,
        "Only object files",
        "All files"
    },
},
```

■ **Figure 13-29** Example of multiple input files with object files specified
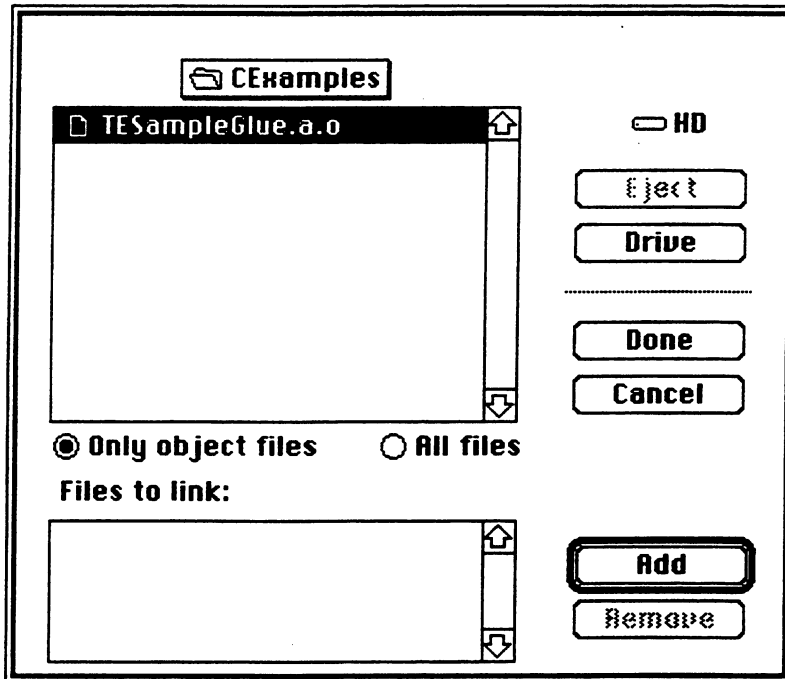
In Figure 13-29, TESampleGlue.a.o is the only file in the CExamples directory that has a type of 'OBJ '. After the "All files" radio button is clicked, all files in the CExamples directory are displayed, as shown in Figure 13-30.

■ **Figure 13-30** Example of multiple input files with all files specified

```
┌─────────────────────────────────────────────┐
│  ┌──────────────────────────────────┐        │
│  │        ⊟ CEHamples                │        │
│  │  ┌──────────────────────────┐┌─┐ │  ⊂⊃HD  │
│  │  │ ▯ TESample.h             ││⇧│ │        │
│  │  │ ▯ TESample.make          ││▒│ │ ┌─────┐│
│  │  │ ▯ TESample.r             ││▒│ │ │Eject││
│  │  │ ▯ TESampleGlue.a         ││▒│ │ └─────┘│
│  │  │ ▮ TESampleGlue.a.o       ││▒│ │ ┌─────┐│
│  │  │ ▯ TestPerf.c             ││▒│ │ │Drive││
│  │  │ ▯ TubeTest.c             ││▒│ │ └─────┘│
│  │  │ ▯ TubeTest.make          ││ │ │ ┌─────┐│
│  │  │ ▯ TubeTest.r             ││⇩│ │ │Done ││
│  │  └──────────────────────────┘└─┘ │ └─────┘│
│  │  ○ Only object files  ◉ All files│┌──────┐│
│  │  Files to link:                  ││Cancel││
│  │  ┌──────────────────────────┐┌─┐ │└──────┘│
│  │  │                          ││⇧│ │ ┌─────┐│
│  │  │                          ││ │ │ │ Add ││
│  │  │                          ││⇩│ │ └─────┘│
│  │  └──────────────────────────┘└─┘ │ Remove │
│  └──────────────────────────────────┘        │
└─────────────────────────────────────────────┘
```
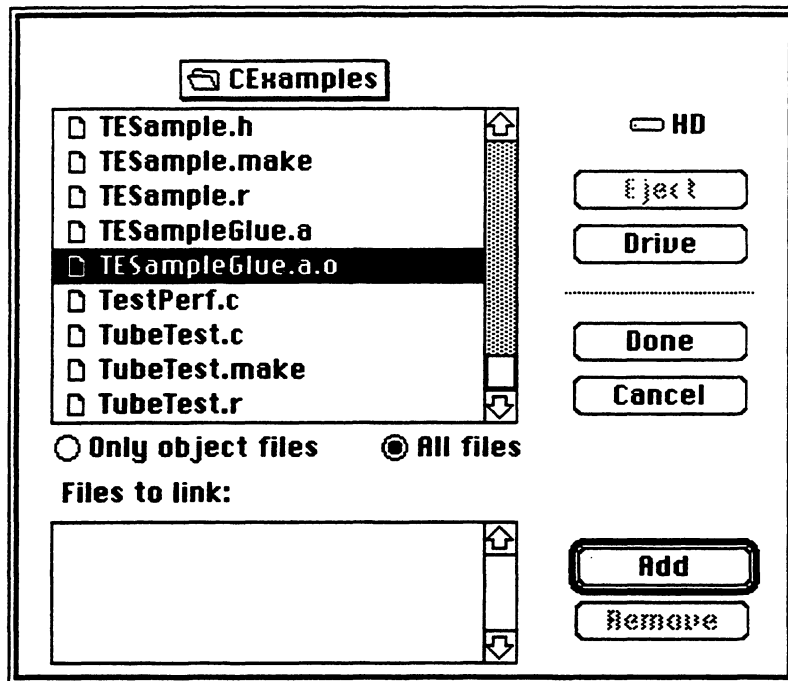
Multiple files and directories for input only

Here's how you can use the case MultiFiles to enable the user to select multiple directories for input only.

```
NotDependent {}, MultiFiles {
    "Include Paths...",
    "Help message for directory button.",
    {110, 330, 130, 468},
    "Include Search Paths:",
    "-s",
    MultiDirs {},
},
```

The first item in the above code, "Include Paths...", is the button in a frontmost dialog box (Rez was used in this example) that generates the file dialog box shown in Figure 13-31. "Include Search Paths:" is the title of the scrollable window at the bottom of the dialog box. Two Includes folders have just been selected from the upper window and added to the Include Search Paths window just below.

■ **Figure 13-31** Multiple directories for input
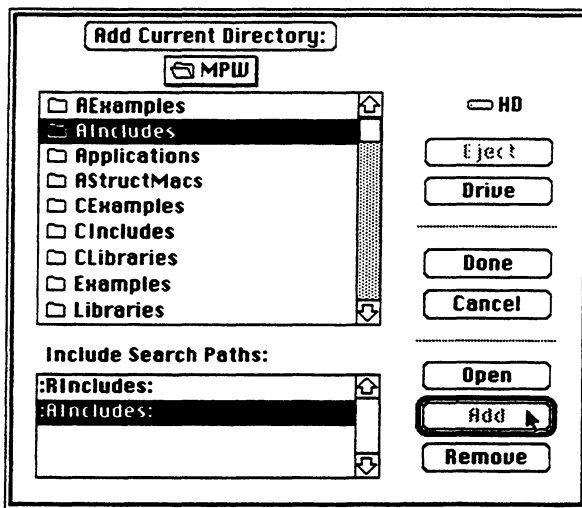
Another file dialog box is used to select multiple files and directories. This dialog box appears in Figure 13-32. Here is the resource code that produces this dialog box.

```
NotDependent {}, MultiFiles {
        "Files to duplicate…",
        "This button brings up a dialog allowing"
            "selection of files and directories to duplicate.",
        {25, 50, 45, 230},
        "Files and Directories to duplicate:",
        "",
        MultiInputFilesAndDirs {}
},
```

■ **Figure 13-32** Example of a "directories" control for multiple input files
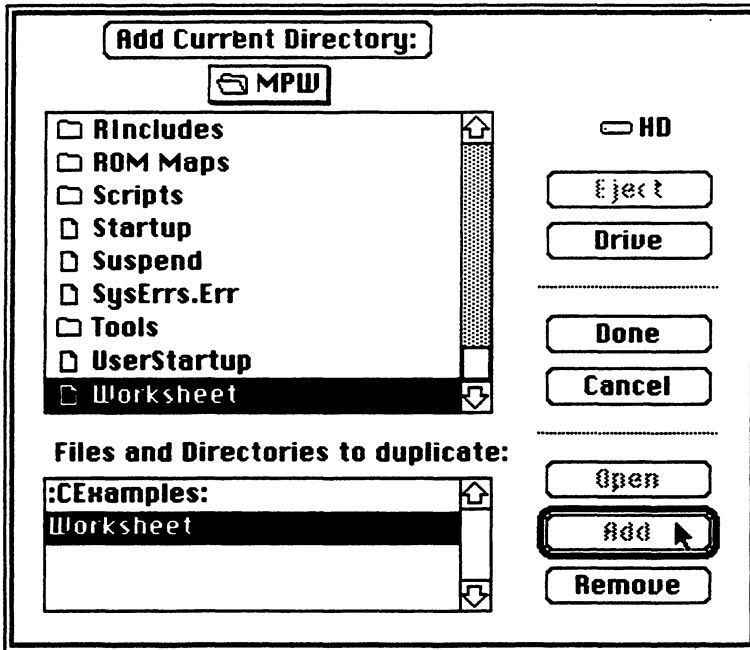
Multiple new files

The case MultiFiles also gives the user the ability to select multiple files for output.

Here is the resource code resulting in the example shown in Figure 13-33.

```
notDependent { }, MultiFiles {
    "New Files...",
    "Help message for button",
    {110, 330, 130, 468},
    "New files to open:",
    "-n",
    MultiOutputFiles { },
};
```

■ **Figure 13-33** Using the MultiOutputFiles subcase of the case MultiFiles
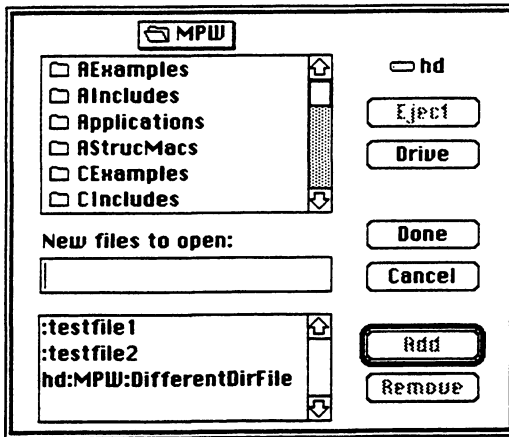
Version

You can place a version string in your Commando dialogs for your own identification purposes, as shown in Figure 13-34. The version string is centered below the Do-It button. Here is the declaration for VersionDialog:

```
case VersionDialog:                    /* Display a dialog when the version #
                                          is clicked */
        key byte = VersionDialogID;
        switch {
                case VersionString:    /* Version string embedded right here */
                        key byte = 0;
                        cstring;        /* Version string of tool (e.g. V2.0) */


                case VersionResource:  /* Versions string comes from another
                                          resource */
                        key byte = 1;
                        literal longint; /* resource type of pascal string
                                            containing version string */
                        integer;        /* resource id of version string */
        };
        cstring;                        /* Version text for help window */
        align word;
        integer   noDialog;            /* Rsrc id of 'DLOG' */
```

If there is no modal dialog to display when the version string is clicked, set the resource ID
to zero (noDialog).

If the version string comes from another resource (VersionResource), the string must be
the first thing in the resource, and the string must be a Pascal-style string. An `STR `
resource is an example of a resource that fits the bill.

If the modal dialog is to have a filter procedure, the procedure must be linked as an
`fltr` resource with the same resource ID as the dialog.

■ **Figure 13-34**   Version string
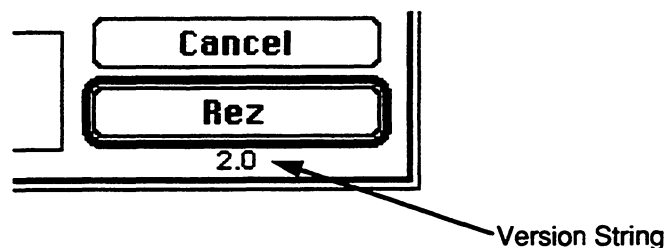
Version String

The version string may be embedded in the commando resource using the VersionString case or the version string may come from a resource using the VersionResource case. If the version comes from a resource, the resource must contain a Rez-style `pstring`. You can use this with the SetVersion tool to read SetVersion's `'MPST'` resource.

As usual, the help string is a string that is displayed when the version string is clicked. Typically, this help string contains more detailed author/version information.

For extra flair, a dialog may be zoomed out when the version string is clicked. If a dialog is specified, you must give the resource ID of the `'DLOG'` resource (found in the resource fork of your tool or script) to display. Commando simply calls ModalDialog( ) with that dialog.

If you want to have a custom filter procedure, you must compile the filter procedure as a standalone resource with a resource type of `'fltr'` and with the same id as the `'DLOG'` resource. The visible/invisible flag in the DLOG resource should be set to invisible. Commando will move the `'DLOG'` window so that the bounds rect specified in the `'DLOG'` are relative to the bounds of the Commando dialog.

◆ *Note:* If you do not specify a VersionDialog commando item, Commando attempts to add one for you by looking for a 'vers' resource with an ID of 1. If found, Commando displays the short version string under the Do-It button. When the version string is clicked, Commando displays the long version string in the help window. If a 'vers' (1) resource is not found, Commando looks for a 'vers' (2) resource. If one is not found, no version string is displayed.

A Commando example

The best way to learn how to make a Commando interface is to study an actual Commando resource for an existing MPW tool. Choose a tool, explore the operation of the controls in its Commando dialog, and then use DeRez to generate a readable version of the tool's Cmdo.r resource.

To obtain the Commando resource for a tool, use this syntax:

```
DeRez {MPW}Tools:toolname Cmdo.r -only cmdo
```

To obtain the Commando resource for a Shell command, use this syntax:

```
DeRez "{MPW}MPW Shell" Cmdo.r -only "'cmdo' (∂"Commandname∂")"
```

For your convenience, the Commando resource for ResEqual, called ResEqual.r, is shown here. You can find this file in the PExamples folder.

```
#include "cmdo.r"
resource 'cmdo' (355) {
    {
        240,
        "ResEqual compares the resources in two files and reports
         the differences.",
        {
            NotDependent {}, Files {
                InputFile,
                RequiredFile {
                    {40, 40, 60, 190},
                    "Resource File 1",
                    "",
                    "Select the first file to compare.",
                },
                Additional {
                    "",
                    FilterTypes,
                    "Only applications, DA's, and tools",
                    "All files",
                    {
                        appl,
                        tool,
                        da
                    }
                }
            },
            Or {{1}}, Files {
                InputFile,
                RequiredFile {
                    {70, 40, 90, 190},
                    "Resource File 2",
                    "",
                    "Select the second file to compare.",
                },
                Additional {
                    "",
                    FilterTypes,
                    "Only applications, DA's, and tools",
                    "All files",
                    {
                        appl,
```

```
                         tool,
                         da
                    }
               }
     },
     NotDependent {}, TextBox {
          gray,
          {30, 35, 95, 195},
          "Files to Compare"
     },
     NotDependent {}, CheckOption {
          NotSet,
          {105, 75, 121, 155},
          "Progress",
          "-p",
          "Write progress information to diagnostic"
          "output."
     },
     NotDependent {}, Redirection {
          StandardOutput,
          {40, 300}
     },
     NotDependent {}, Redirection {
          DiagnosticOutput,
          {80, 300}
     },
     NotDependent {}, TextBox {
          gray,
          {30, 295, 121, 420},
          "Redirection"
     },
     Or {{2}}, DoItButton {
     },
          }
     }
};
```

The above resource code generates the frontmost dialog box of ResEqual, which appears in Figure 13-35.

■ **Figure 13-35** A Commando example: frontmost ResEqual dialog box
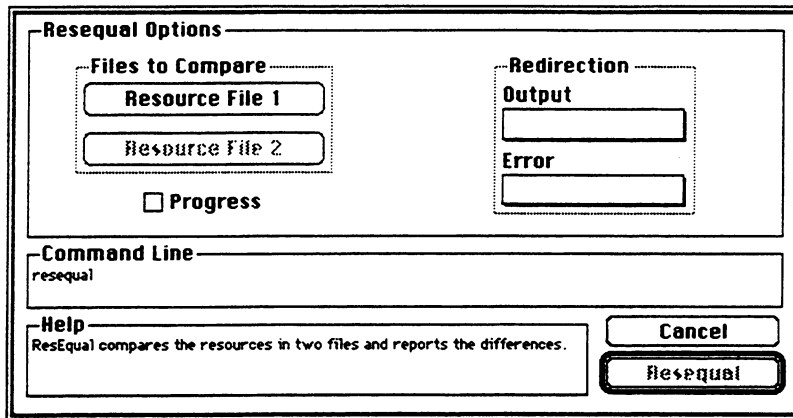
Chapter 14 **Performance-Measurement Tools**

MPW 3.0 PROVIDES A SET OF PERFORMANCE-MEASUREMENT TOOLS to aid professional software developers in measuring and improving the performance of their applications. This chapter explains how to use these tools and provides a detailed example. The PerformReport tool is also described in Part II. ∎

Contents

About performance-measurement tools

In essence, the performance-measurement tools sample the Program Counter (PC) register just often enough to obtain a statistically accurate estimate of the program's actual use of time. The code is divided into "buckets" of two or more bytes and a count of sampled PC values for each bucket during the program's execution is output to a text file. You can then analyze these results by running a report generator, PerformReport. PerformReport merges the output file with a linkmap of the measured code resources to produce a list of procedures, sorted by the number of PC samples found within the procedure.

▲ **Warning** The performance-measurement tools are designed for temporary inclusion in an application, desk accessory, or driver for purposes of measuring performance. They are not designed for inclusion in commercial products, because they rely on low-level system mechanisms that are not guaranteed to function correctly on all future machines. ▲

The memory management strategy for the performance tools is based on the assumption that developers wishing to measure performance will likely have a machine larger than the smallest target machine for their applications. Thus, they can use performance tools that require some additional memory without severely impacting the application's memory management strategy.

The best way to use these tools depends upon your particular environment and the code you want to test. These considerations are discussed in the section "Implementation Issues" later in this chapter. You will need to temporarily insert calls to the performance tools within your code. Examples of the placement of these calls are provided in MPW C and MPW Pascal. Be sure to remove these calls when you have completed your optimizations.

Components of performance tools

The performance tools consist of the following pieces:

- **A library file (PerformLib.o):** This file is in the {Libraries} folder. Link with this file.

- **Interface files for Pascal (Perf.p) and C (Perf.h):** These files are in the interface folders {PInterfaces} for Pascal and {CIncludes} for C. These are the files that you use or include in the source files for your application. These interfaces depend only on the standard Macintosh memory types files: MemTypes.p for Pascal and Types.h for C.

  An assembly-language interface has not been provided for the performance tools. Assembly-language programmers can use either the Pascal or the C interface. Both go directly to the Pascal and assembly-language implementation in PerformLib.o.

- **Sample programs, makefiles, and instructions for execution:** These files are in the Examples folders: {MPW}Examples:PExamples: for Pascal, and {CExamples}Examples:CExamples: for C. Instructions for running the performance tools sample programs are included in the Examples folders.

- **PerformReport (a tool for analyzing performance data and producing reports):** This tool is found in the {MPW}Tools: folder. For detailed information about the tool, see the command pages in Part II. For detailed instructions on how to run this tool, see the instructions in the appropriate Examples folder. Examples of the output from this tool are discussed below.

- **ROM map files:** You'll find a number of ROM maps in the folder {MPW}'ROM Maps', including MacIIROM.map, MacSEROM.map, and MacPlusROM.map. These files are combined with the link map file for your application, to add location information for the OS and Toolbox routines to the performance data. You will usually append the appropriate ROM maps to your application's link map for input to the tool PerformReport.

Requirements for using performance tools

To use the performance tools, you need to add calls to these routines in your application, desk accessory, or driver. They are described later in more detail:

■ **InitPerf** specifies the types of measurements to be performed, and allocates storage. This should be called once near the beginning of your code.

■ **TermPerf** stops measurements (if active), and frees the storage. `TermPerf` must be called once after `InitPerf` succeeds, and measurement is finished.

■ **PerfControl** starts and stops measurements. `PerfControl` must be called once (after `InitPerf`) to start measurements. Use `PerfControl` to avoid taking measurements in idle loops, dialog boxes, alerts, and other places where the user response time determines performance.

■ **PerfDump** stops measurements (if active), and writes the performance data to an output file. You should call `PerfDump` after measurements are collected for reporting.

How performance measurement works

The performance-measurement tools are designed to give you useful information about the performance of a program without severely altering the user responsiveness or memory requirements—that is, without changing the characteristics of what is being measured. However, the act of measurement necessarily alters what is being measured in the ways summarized below.

Program Counter sampling

The fundamental idea behind the performance-measurement tools is to sample the Program Counter (PC) register frequently enough to obtain a statistically accurate estimate of the actual program performance, but infrequently enough so that overall performance is not affected. The performance-measurement tools use the Vertical Blanking signal (VBL) on 64K ROMs and the Time Manager on 128K and larger ROMS.

The Time Manager allows 1 ms resolution in sampling, but this imposes about a 20 percent performance degradation. A value of 4 ms to 10 ms reduces the performance degradation to 4 percent to 10 percent. Use of the VBL signal on old ROMs imposes a sampling rate of approximately 60 times per second (16 ms).

Restrictions

If your application directly uses the VIA Timer1 (or some software that uses it, such as the sound generator or the Time Manager) then you might not be able to use these performance-measurement tools.

In the case of old ROMs, the performance-measurement tools may not work correctly with programs that make use of VBL tasks.

If you are running the performance tools under MultiFinder, you may need to increase the sampling interval.

▲ **Warning**   If you set the sampling interval too low for your machine, the performance tools may crash or cause your program to run very slowly. It is best to start with a high sampling interval, such as 10 ms or 20 ms, and decrease it only after experience allows you to predict the effect of the shorter interval. For example, if measurements taken with a sampling rate of 10 ms cause your program to run 10 percent slower, then it is probably safe to increase the sampling rate to every 5 ms at a cost of having the program run 20 percent slower. ▲

Bucket counts

The performance tools require 2 bytes of memory for a counter for each "bucket" of code that is measured. For instance, for a 100K tool or application, using a bucket size of 16 bytes, about 12,800 bytes are required for the counters. If the ROM is measured, an additional 8K, 16K, or 32K bytes (for 64K, 128K, or 256K ROMs) is required.

If your program spends a substantial amount of time outside CODE segments and ROM, then you may want to measure RAM "misses." Because RAM can be quite large, a second (generally larger) bucket size can be specified for RAM "misses." And you can control the amount of RAM to be measured by using a low address to start setting up buckets and a high address for the last bucket. If the RAM misses are measured, additional memory is required.

The sum of all memory required for counters is allocated as a single contiguous block at the time InitPerf is called. For this reason, you should call InitPerf fairly early in your initialization, before memory becomes fragmented.

In addition to the memory for bucket counters, the performance tools will use one master pointer for a handle to some information, and will allocate a few small structures with NewPtr calls.

Using performance-measurement tools

This section presents a detailed explanation for each of the seven steps necessary to install the performance-measurement routines into your code. For each step the specifics for using these tools with MPW C and MPW Pascal are under separate subheadings.

You need make only a few changes to install these tools in your code. The changes are basically the same, whether you are developing an application, a desk accessory, an MPW tool, or a driver. It is even possible to install performance tools in ROM.

Here are the steps:

1. Install under conditional compilation

After measuring the performance of your program, you will probably want to make changes, test the changes for correctness, and then repeat the measurements to verify the performance improvements. While making and testing changes, it is very important not to include the performance tools, unless you are confident that the changes do not introduce any new bugs. If your code terminates early for any reason, then the normal system recovery techniques (in MacsBug, calls such as G SysRecover under the MPW Shell or ES from an application) do not work. In such a case, within a few milliseconds after the system tries to reuse the memory occupied by the performance tools, a timer interrupt occurs and a system error or crash results. The system error will probably force rebooting the system. For this reason, it is advisable to include the performance-measurement tools under a conditional flag.

◆ Note: In the steps that follow, it is assumed that all the performance measurement changes are surrounded by conditional compilation. However, in the code fragments that follow, the actual conditional compilation statements are omitted to save space.

MPW C

```
/*
        #define PERFORMANCE to turn on the measuring tools.
        #undef PERFORMANCE to turn off the measuring tools.
*/
#define PERFORMANCE
```

Calls to the performance tools routines can then be surrounded by the following conditional compilation statements:

```
#ifdef PERFORMANCE
...
#endif PERFORMANCE
```

MPW Pascal

```
{$SETC DoPerform := true}   {false to exclude Performance Tools}
```

Calls to the performance tools routines can then be surrounded by the following conditional compilation statements:

```
{$IFC DoPerform}
...
{$ENDC}
```

---

2. Include the interface

In the main body of your MPW C code, you need to include the header file for the performance tools, like this:

```
#include <Perf.h>
```

In the main body of your MPW Pascal code, you need to reference the interface file for the performance tools, like this:

```
USES
        MemTypes,
        Perf;
```

3. Provide a pointer to a block of variables

For an application or MPW tool, you can declare a global variable. If you are developing a desk accessory, driver, or ROM that does not have global variables, then you need to be somewhat creative in finding a location for the pointer. The choices include: a local variable on the stack (assuming the stack frame will persist long enough), a field of a block allocated and locked down in the heap, or a low memory location. In any event, the address of the location allocated for the pointer must be passed to the performance routines, as indicated in the following steps.

MPW C

```
TP2PerfGlobals ThePGlobals;
```

MPW Pascal

```
VAR thePerfGlobals: TP2PerfGlobals;
```

4. Initialize the performance-measurement tools

Somewhere near the beginning of your code's execution, and when large chunks of memory are available, you need to initialize the performance tools.

▲ **Warning**   Once your code has initialized the performance routines successfully, it is important that you call the termination routine described in Step 7 before your code terminates. Failure to do so almost always results in a fatal system crash. ▲

MPW C

```
ThePGlobals = nil;
if (!InitPerf(&ThePGlobals, ... other parameters ...)) {
        /* report error in initialization and terminate */
    };
```

The function InitPerf allocates a block on the heap for the performance global variables if ThePGlobals is nil. If the ThePGlobals is not nil, InitPerf assumes the block is already allocated.

MPW Pascal

```
thePerfGlobals := NIL;
IF NOT InitPerf(thePerfGlobals,  .. other parameters ...) THEN
      BEGIN
      {Report error in initialization and terminate.}
      END;
```

When you set the pointer `thePerfGlobals` to `NIL`, `InitPerf` allocates a block on the heap for the performance global variable. If the pointer is not `NIL`, `InitPerf` assumes the block is already allocated.

---

5. Turn on the measurements

After initialization succeeds, you can start measurements at any point in your code. The call that starts (and stops) measurements returns the current on-off state as a Boolean value.

You can call `PerfControl` with a second argument of `false` in order to turn performance measurements off. This is useful for disabling sections of code that you don't want to measure, such as the event loop of an application, a dialog box where user response time dominates the compute time, parts of the application that rely on the VIA timer, and so on.

MPW C

```
(void) PerfControl (ThePGlobals, true);
```

MPW Pascal

```
VAR OldState: boolean;
...
OldState := PerfControl (thePerfGlobals, true);
```

Alternatively, you may use:

```
IF PerfControl(thePerfGlobals, true)
      THEN {dummy THEN statement};
```

6. Dump the results

When you reach the end of the code to be measured, you must make a call to have the performance counters written into a text file. If the dump routine encounters any I/O, memory management, or other system errors, it returns a nonzero return code. You can examine this code to determine the nature of the problem.

MPW C

```
OSErr err;

...
err = PerfDump(ThePGlobals, "\pPerform.out", ...other parameters);
if (err != noErr)
        /* Code to report erros during dump */
```

The `PerfDump` routine takes the output-filename as a Pascal string. If the empty string is passed, the name defaults to Perform.out.

MPW Pascal

```
VAR err: OSErr;

...
err := PerfDump(thePerfGlobals, 'Perform.out', ...other parameters);
If err <> noErr
        THEN {Report errors during dump};
```

If the empty string is passed for a filename, the name will default to Perform.out.

7. Terminate cleanly

After dumping the counters to a text file, you must terminate the performance-measurement tools cleanly. `TermPerf` removes the interrupt routine and frees the memory associated with the performance global variables and counters.

MPW C

```
TermPerf(ThePGlobals);
```

MPW Pascal

```
TermPerf(thePerfGlobals);
```

MPW performance tools routines

This section gives detailed information about MPW C and MPW Pascal parameters to the performance tools routines. The C and Pascal calls are presented first, followed by discussion relevant to both.

The function InitPerf

Here is the MPW C declaration for InitPerf:

```
pascal Boolean InitPerf(
        TP2PerfGlobals          *thePerfGlobals,
        short                   timerCount,
        short                   codeAndROMBucketSize,
        Boolean                 doROM,
        Boolean                 doAppCode,
        const                   Str255 appCodeType,
        short                   romID,
        const                   Str255 romName,
        Boolean                 doRAM,
        long                    ramLow,
        long                    ramHigh,
        short                   ramBucketSize
);
```

Here is the MPW Pascal declaration for InitPerf:

```
FUNCTION InitPerf (
        VAR thePerfGlobals:             TP2PerfGlobals;
        timerCount, codeAndROMBucketSize:   integer;
        doROM, doAppCode:               boolean;
        appCodeType:                    Str255;
        romID:                          integer;
        romName:                        Str255;
        doRAM:                          boolean;
        ramLow, ramHigh:                longint;
        ramBucketSize:                  integer
) : boolean;
```

Call the function `InitPerf` once to set up the performance-monitoring interrupt handler and to allocate the memory area for counters. The function returns `true` if initialization is successful, and `false` if it encounters errors.

The function `InitPerf` takes a number of parameters:

- `thePerfGlobals` is the address of the pointer to the global variable area. If the value of the pointer is `nil`, a new block of global variables is allocated on the heap.

- `timerCount` (for new ROMs) determines the number of milliseconds between PC samples. For most applications, good values are:

  - 10 ms for Macintosh Plus and Macintosh SE, when running under the Finder. Under MultiFinder, allow 20 ms.

  - 4 ms for Macintosh II, running under the Finder. Under MultiFinder, allow 10 ms.

  - ◆ *Note:* For old (64K) ROMs, `timerCount` is the number of VBL events (16 ms each) between PC samples.

- `codeAndROMBucketSize` sets the bucket size for user code (and the ROM, if ROM measurement is requested). A separate parameter sets the bucket size for RAM, as described below. The bucket size may be any integer greater than or equal to 2.

  - ◆ *Note:* The performance tools force the bucket size to be a power of 2 by rounding this parameter up to the nearest power of 2.

If the bucket size is set as low as 2, individual instructions are measured. However, this requires a lot of memory—an amount equal to the amount of code (and ROM) being measured.

A more practical value for this parameter is 8, which requires only 25 percent of the memory being measured. Even larger bucket sizes may be used if memory is scarce, although the resolution of the measurements becomes an issue at some point.

- `doROM` determines whether the ROM code as well as the user's code are measured. A value of `true` causes the ROM code to be measured.

- `doAppCode` determines whether or not user code is measured. A value of `true` causes user code to be measured.

- `appCodeType` is a Pascal string that determines the resource type of user code to be measured. For application programs this should be `'CODE'` (in Pascal) or `"\pCODE"` (in C); for desk accessories it should be `'DRVR'` (in Pascal) or `"\pDRVR"` (in C); and so on. Resources of the specified type are obtained from the current (top-level) resource file.

- `romID` indicates ROM types. You'll normally pass a `romID` of 0, indicating the use of one of the predefined ROMs. Table 14-1 shows the predefined ROM IDs and names.

Computer	ROM ID	ROM name
Macintosh 128K	$69	ROM
Macintosh XL	$FF	ROMXL
Macintosh Plus	$75	ROMPLUS
Macintosh 512e	$75	ROMPLUS
Macintosh SE	$76	ROMSE
Macintosh II	$78	ROMII
Macintosh IIx	$78	ROMII (unchanged from Macintosh II)

ROM IDs and the following parameters are mainly to support older or newer ROMs not in Table 14-1.

■ `romName` indicates a ROM name other than one of the predefined names listed in Table 14-1. This value is usually the empty string, indicating the use of a predefined ROM name. This parameter can be used to specify the name of older or newer ROMs.

■ `doRAM` determines whether RAM misses are measured. A value of `true` invokes measurement.

■ `ramLow` specifies the lower limits of RAM to measure for misses. This parameter has no effect unless `doRAM` is `true`.

■ `ramHigh` specifies the upper limit of RAM to measure for misses. This parameter has no effect unless `doRAM` is `true`.

■ `ramBucketSize` specifies the bucket size to use for measuring RAM misses. This parameter has no effect unless `doRAM` is `true`.

A "RAM miss" is a PC sample that is not contained in any of the user code segments or the ROM.

The function PerfControl

Here is the MPW C declaration for `PerfControl`:

```
pascal Boolean PerfControl(
     TP2PerfGlobals thePerfGlobals,
     Boolean turnOn
);
```

Here is the MPW Pascal declaration for `PerfControl`:

```
FUNCTION PerfControl(
        thePerfGlobals: TP2PerfGlobals;
        turnOn: boolean
) : boolean;
```

The `PerfControl` function returns the previous state. You must call `PerfControl` once to begin performance measurements. It can be called more frequently to avoid measuring uninteresting areas of code, such as idle loops or dialog boxes.

■ `thePerfGlobals` points to the global variable area, initialized by a successful call to `INITPERF`.

■ `turnOn` turns measurements on (`true`) and off (`false`).

The function PerfDump

Here is the MPW C declaration of `PerfDump` :

```
pascal short PerfDump(
        TP2PerfGlobals      thePerfGlobals,
        const Str255        reportFile,
        Boolean             doHistogram,
        short               rptFileColumns
);
```

Here is the MPW Pascal declaration of `PerfDump`:

```
FUNCTION PerfDump(
        thePerfGlobals:    TP2PerfGlobals;
        reportFile:        Str255;
        doHistogram:       boolean;
        rptFileColumns:    integer
): integer{OSErr};
```

The function `PerfDump` dumps the statistics gathered by the performance tools into a text file suitable either for direct analysis or for processing by PerformReport. `PerfDump` calls `PerfControl` to turn off measurements and accepts the following parameters:

■ `thePerfGlobals` points to the global variable area, initialized by a successful call to `InitPerf`.

■ `reportFile` specifies the name of the report file. If this is the empty string, the default name Perform.Out is used.

- `doHistogram` (if `true`) places a histogram after the bucket counts in the data file. The histogram consists of a number of asterisks for each bucket, normalized so that the bucket with the largest number of hits receives a line of asterisks out to `rptFileColumns`.

- `rptFileColumns` controls the number of columns in the report file. It has no effect unless `doHistogram` is `true`.

The function TermPerf

Here is the MPW C declaration of `TermPerf` :

```
pascal void TermPerf(TP2PerfGlobals thePerfGlobals);
```

Here is the MPW Pascal declaration of `TermPerf`:

```
PROCEDURE TermPerf(thePerfGlobals: TP2PerfGlobals);
```

If the call to `InitPerf` succeeds, then `TermPerf` must be called before terminating the program. Otherwise, a system crash results because the timer interrupt, which is still enabled, will jump to points unknown. `TermPerf` removes the interrupt handler and frees the storage used by the counters with the parameter `thePerfGlobals`. This parameter points to the global variable area, initialized by a successful call to `InitPerf`.

Performance reports

When your code has completed its execution, you call `PerfDump` to generate a performance output file showing the results of the bucket counts. You can analyze this data by using the tool `PerformReport`. Examples of both the performance output and report files appear in this section. See Part II for a command page describing the tool `PerformReport`.

Performance output file

The results of the performance tests are output to a performance data file when `PerfDump` is called. This file is a text file containing the bucket locations and counts. You should call `PerfDump` at the very end of the test, so that no interference with program I/O should occur. The performance output file is not opened until `PerfDump` is called.

Below is an example of a performance output file as generated by a call to `PerfDump`. Some repeated lines have been omitted, as indicated by "...".

Notice that the performance data is arranged on a per segment basis. Only nonzero buckets are reported; in other words, missing buckets had a hit count of zero. (`PerfDump` has an option to produce a histogram (bar graph) to the right of the Hits column. That option was not exercised in this example.)

```
Performance Parameters
======================


Bytes per bucket, Code and ROM: 8
Bytes per bucket, RAM: 4
Sampling Interval: 4 ms

Performance Summary
===================


Total hits outside of the sampled segments: 2
Maximum hits in one bucket: 872
Total hits in all buckets: 3222


Performance Data
================


Offset Hits | Segment 117   size   20000
========================================================================
   52F8      1 |
   5300      1 |
   53C8     12 |
   53E8      1 |
   53F0      2 |
   5400      1 |
   5428      1 |
   5728    872 |
   ...
  1B830      9 |
  1B838     53 |
  1B840     41 |
  1B848     61 |
  1B850     41 |


Offset Hits | Segment 253   size 1FFFFF
========================================================================
   D6A0      1 |
  287D0     40 |
  1E6134     1 |
  1E8990     1 |
  1FB20C   101 |

Offset Hits | Segment 13   size    B68   name STDIO
========================================================================


Offset Hits | Segment 12   size    71E   name SACONSOL
========================================================================


   ...
```

```
Offset Hits | Segment 5   size      D0   name ROMSEG2
================================================================
   70     2 |
   88     5 |
   90    10 |
...
   B0     4 |
   B8     2 |
   C0     3 |

Offset Hits | Segment 4   size     136   name ROMSEG1
================================================================
   10     9 |
   18     3 |
   20     5 |
...
  110    19 |
  118    58 |
  120    46 |

Offset Hits | Segment 3   size      8C   name SEG2
================================================================
   50     3 |
   58     3 |
   60     9 |
   68     1 |
   70    14 |
   78     1 |

Offset Hits | Segment 2   size      D0   name SEG1
================================================================
   10     2 |
   18    18 |
   20     4 |
...
   A8     7 |
   B0    12 |
   B8    18 |

Offset Hits | Segment 1   size    101C   name Main
================================================================
  F38    43 |
  F40   116 |
  F48    78 |
  F50    19 |
  F58    77 |
  F60    66 |
  F68    56 |
```

Analyzing the results with PerformReport

Once the performance data file has been generated, you are ready to run the report generator, a tool called `PerformReport`. This tool merges the performance output file with a linkmap of the measured code resources to produce a list of procedures, sorted by the number of PC samples found within the procedure. (See Part II for more information on `PerformReport`.) An example of the contents of this file is shown here.

If your call to `InitPerf` had the parameter `doRom` set to true, then you'll need to append the correct ROM map file to your application's link map before running `PerformReport`. For example:

```
Link -o YourApp -l >LinkMap YourApp.p.o …etc…
YourApp        #  run your application, generate Perform.Out
Catenate {MPW}'ROM Maps':romName.Map >> LinkMap
PerformReport -l LinkMap -m Perform.Out


PerformReport -- Merges Linker Output and Performance Dump January 30,
1989

Reading Link Map file: "LinkMap"

Reading Performance Measurements file: "Perform.Out"

PerformReport Parameters:

      8 bytes per bucket, ROM and CODE.
      4 bytes per bucket, RAM.
      2 hits outside code measured.
   3224 hits total,       0.0% outside the segments.
    872 maximum hits in one bucket.
```

```
Procedures by possible hits (showing Probable % of time):
Num Segment  : Procedure        Def   Prob   Poss   Prob%

117    Main :     ATRAP68020    497    436    872    28.9%
117    Main :       CHKSLOT       0    436    872    13.5%
117    Main :       DSPATCH       0      0    872     0.0%
117    Main :         RSECT     474     13     26    15.1%
  1    Main :       %I_MUL4     399     14     56    12.8%
...
..1    Main :       %I_DIV4       0      3      5     0.6%
  4  ROMSEG1 :       ROMW100       5      0      0     0.1%
117    Main :       LVL1INT       1      0      1     0.0%
117    Main :   TFSDISPATCH       1      0      0     0.0%
117    Main :       LVL2INT       0      0      1     0.0%

        Total Reported =      67.6%  32.1%         99.8%

PerformReport: That's All Folks!
```

Adding identification lines to a data file

After displaying a title line, and giving the names of the files being read, PerformReport
has an option (-e) to echo lines from the head of the measurements file until the phrase
"Performance Data" is encountered. This option allows you to add identification lines at
the head of performance-measurement files. Various parameters are gathered from lines
that begin with special keywords. Here are the keywords with the phrases they head:

Bytes	Bytes per bucket
Total	Total hits
Maximum	Maximum hits
Performance	Performance Data

You are free to add comment lines at the head of a data file, as long as the comment lines
do not begin with these keywords.

Interpreting the performance report

`PerformReport` translates the bucket hit information into procedure-based information. Because procedures can span buckets, there may be some uncertainty about how bucket hits are related to procedure hits. `PerformReport` attempts to deal with this uncertainty by classifying hits into several categories:

Definite   When a bucket is completely contained in a single procedure, all hits in the bucket are counted as definite hits in that procedure.

Possible/Probable   When a bucket is partially contained in several procedures, all hits in the bucket are counted as possible hits in each procedure; in addition, the hits in the bucket are counted as probable hits in a particular procedure, based on the amount of the bucket that is covered by the particular procedure.

Please realize that the concept of probable hits is not intended to give an accurate statistical picture of the situation. What happens in practice is that buckets are frequently covered by two procedures, and almost all of the hits occur in one procedure or the other. The intent behind "possible and probable" hits is to give you some feeling for the accuracy of the resulting data.

If the Pascal example TestPerf.p is modified to have a bucket size of 8, then the possible hits will be few relative to the definite hits. The exception is the `%I_DIV4` procedure, which will have zero definite hits, but shares a bucket with `%I_MUL4`. In fact there are no divide operations in the sample program; therefore all hits apparently belonging to `%I_DIV4` really belong to the multiply operations.

If the percentage of definite hits becomes too low, you should consider reducing the requested bucket size.

Implementation issues

The performance tools have been designed to work "as is" for most common application, desk accessory, and driver runtime environments. However, because Macintosh has an open architecture, it is possible that actions taken or assumptions made by application code will conflict with the needs of the performance tools. This section discusses possible conflicts, and how to resolve them.

Locking the interrupt handler

You must lock down both code and data for the performance tools while taking performance measurements. Code for the trap handler must be locked down because the timer interrupts occur asynchronously. Data for the counters must be locked down because handles cannot be assumed to be valid during interrupt processing. The data area for counters cannot be "grown" at interrupt time, because the heap may be inconsistent.

Segmentation

The code that must be locked down at execution time has been placed in segment "Main" and occupies about 1 kilobyte of space. This is because segment "Main" is usually guaranteed not to be unloaded at run time.

If your application's "Main" segment is too full to allow the performance tools to be linked correctly, then you may retarget the code in PerformLib.o by using the Lib tool. However, your application must not have an "unload all segments" routine in its idle procedure. One good segment to retarget to is "PerfMain", because this segment contains some of the other pieces of the performance tools.

These MPW commands illustrate how to retarget the code in PerformLib.o:

```
Duplicate {Libraries}PerformLib.o temp
Lib -o {Libraries}PerformLib.o -sn Main=PerfMain temp
Delete temp
```

The first command line creates a copy of PerformLib.o in temp. The second line replaces the original PerformLib.o with the output of Lib. The -sn option causes all code originally placed in segment "Main" to be in segment "PerfMain". The third line deletes the file temp.

Dirty CODE segments

Because A5 is not valid at interrupt time, and there are no low memory globals assigned to performance measurement, the interrupt routine stores some data values in its code space, including the pointer to the locked-down data. Thus, if your application uses checksums to detect code segments attacked by errors, the performance tools will cause erroneous checksum failures. The easiest fix is simply not to checksum the "Main" segment (or whichever segment you choose).

Moveable code resources

The code for the trap handler, and the data area for the counters, must be locked down during performance measurement.

In counting "hits" in code resource segments, the performance interrupt routine checks that the handle to a measured resource is locked. If it is not locked, the resource is assumed to be "unloaded" and PC values are not checked for being within the resource.

The performance tools call `stripAddress`, among other A-traps. If you are using A-trap breaks in MacsBug (as with the `ATHC` command), you may get an A-trap from within the Performance Tool's interrupt handler, and MacsBug may state that the heap is corrupt. The heap might not actually be corrupt, but simply inconsistent at interrupt time.

Appendix A Macintosh Programmer's Workshop Files

THIS APPENDIX LISTS ALL OF THE FILES PROVIDED WITH THE *MACINTOSH PROGRAMMER'S WORKSHOP 3.0*. The files are listed as they appear on the distribution disks. (Volume names are shown in bold; directory names begin and end with a colon.) MPW Assembler, MPW Pascal, MPW C, and MPW C++ are separate products. ■

Contents

MPW 3.0 files

Distribution disk MPW Installation Disk:

MPW Installation Disk:

Outside Bug Reporter   Application used to document bugs

MPW Installation Disk:Installation Folder

Backup	Tool used by MPW Installer to copy files
DoIt	Script that shows each file copied
errorFile	File used for error redirection
'MPW Installer'	MPW Shell used for installation procedure
Startup	Script that controls the installation
Worksheet	

Distribution disk MPW1:

MPW1:

'MPW Shell'	The MPW Shell program
MPW.Help	Command syntax descriptions (for Help command)
Quit	Quit MPW script
Resume	Script to resume MPW after executing an application
Startup	Script to initialize MPW Shell
Suspend	Script to suspend MPW to run an application
SysErrs.Err	Indexed error message file (used by Shell and tools)
UserStartup	Customizable startup script called by Startup
Worksheet	Worksheet contents saved from last session

Distribution disk MPW2:

MPW2:Examples:Examples:

AddMenus	Add menu to MPW menu bar
CheckInActive	Check in the active window to Projector
CheckOutActive	Check out the active window from Projector
DerezPict	Derez a PICT data file
Instructions	
Lookup	
'Startup, etc.'	
State	
'Unix Aliases'	

MPW2:Examples:Projector Examples:Sample:

ProjectorDB	Projector database

MPW2:Examples:Projector Examples:Sample:Commands:

ProjectorDB	Projector database

MPW2:Examples:Projector Examples:Sample:Utilities:

ProjectorDB	Projector database

MPW2:Interfaces:RIncludes:

Cmdo.r	Commando graphic interface resources
MPWTypes.r	MPW-specific resource type definitions
Pict.r	Resource type definition for PICT
SysTypes.r	System resource type definitions
Types.r	Common resource type definitions

MPW2:Libraries:Libraries:

DRVRRuntime.o	Driver runtime library
HyperXLib.o	Libraries for Hypercard XCMD's and XFCN's
Interface.o	*Inside Macintosh* interface library
ObjLib.o	Object-oriented programming library
PerformLib.o	Library for performance-measurement tools
Runtime.o	Runtime library for Assembler and Pascal
SERD	Serial driver resources
Stubs.o	Stub routines to make MPW tools smaller
ToolLibs.o	MPW tool library (spinning cursor, error manager)

MPW2:ROM Maps:

MacIIROM.map
MacPlusROM.map
MacSEROM.map

MPW2:Scripts:

BuildCommands	Automated build commands
BuildMenu	Generates menu for use with automated build commands
BuildProgram	Automated build
CCvt	Converts 2.0 C source to 3.0
CompareFiles	Compares two files side by side
CompareRevisions	Compares two revisions of the same file
CreateMake	Generates a makefile to build a program
DirectoryMenu	Generates Directory menu
DoIt	Highlights and executes a series of Shell commands
Line	Locates line number (useful with other tools and scripts)
MergeBranch	Merges a branch revision onto a project's trunk
OrphanFiles	Removes Projector information from files
SetDirectory	Command to set current directory
TransferCKID	Move Projector information from one file to another
UserVariables	Use Commando to set all user variables

Distribution disk MPW3:

MPW3:Tools:

AboutBox
Backup
Canon
Canon.Dict
CCvtMxL.dict
CCvtUMx.dict
Choose
Commando
Compare
Count
DeRez
DumpCode
DumpFile

DumpObj
Entab
FileDiv
GetErrorText
GetFileName
GetListItem
Lib

Distribution disk MPW4:

MPW4:Tools:

Link
Make
MakeErrorFile
MatchIt
PerformReport
Print
ProcNames
ResEqual
Rez
RezDet
Search
SetPrivilege
SetVersion
Sort
Translate
WhereIs

MPW Assembler files

Distribution disk MPW Assembler1:

MPW Assembler1:Examples:AExamples

Count.a
Count.r
FStubs.a
Instructions
MakeFile
Memory.a
Sample.a
Sample.h
Sample.incl.a
Sample.make
SampleMisc.a
Sample.r

MPW Assembler1:Tools

Asm

Distribution disk MPW Assembler2:

MPW Assembler2:Interfaces:AIncludes:

ApplDeskBus.a
ATalkEqu.a
FixMath.a
FSEqu.a
FSPrivate.a
Graf3DEqu.a
HardwareEqu.a
HyperXCmd.a
IntEnv.a
ObjMacros.a
PackMacs.a
PaletteEqu.a
PickerEqu.a
PrEqu.a

PrintCallsEqu.a
PrintTrapsEqu.a
Private.a
PrPrivate.a
QuickEqu.a
ROMEqu.a
SANEMacs.a
SANEMacs881.a
ScriptEqu.a
SCSIEqu.a
ShutDownEqu.a
Signal.a
SlotEqu.a
SonyEqu.a
Sound.a
SysEqu.a
SysErr.a
TimeEqu.a
ToolEqu.a
Traps.a
VideoEqu.a

MPW Assembler2:Interfaces:AStructMacs:

FlowCtlMacs.a
ProgStrucMacs.a
Sample.a
Sample.r

MPW Pascal files

Distribution disk MPW Pascal1:

MPW Pascal1:Libraries:PLibraries:

PasLib.o
SANELib.o
SANELib881.o

MPW Pascal1:Tools:

Pascal
PasMat
PasRef

Distribution disk MPW Pascal2:

MPW Pascal2:Examples:PExamples:

EditCdev.make
EditCdev.p
EditCdev.r
FStubs.a
Instructions
MakeFile
Memory.p
Memory.r
ResEqual.p
ResEqual.r
Sample.h
Sample.make
Sample.p
Sample.r
SillyBalls.make
SillyBalls.p
TESample.h
TESample.make
TESample.p
TESample.r
TESampleGlue.a
TESampleGlue.a.o
TestPerf.p
TubeTest.make
TubeTest.p
TubeTest.r

MPW Pascal2:Interfaces:PInterfaces:

AppleTalk.p
Controls.p
CursorCtl.p
Desk.p
DeskBus.p

Devices.p
Dialogs.p
DisAsmLookUp.p
DiskInit.p
Disks.p
ErrMgr.p
Errors.p
Events.p
Files.p
FixMath.p
Fonts.p
Graf3D.p
HyperXCmd.p
IntEnv.p
Lists.p
MacPrint.p
Memory.p
MemTypes.p
Menus.p
Notification.p
ObjIntf.p
OSEvents.p
OSIntf.p
OSUtils.p
Packages.p
PackIntf.p
PaletteMgr.p
Palettes.p
PasLibIntf.p
Perf.p
Picker.p
PickerIntf.p
Printing.p
PrintTraps.p
Quickdraw.p
Resources.p
Retrace.p
ROMDefs.p
SANE.p
Scrap.p
Script.p
SCSI.p
SCSIIntf.p
SegLoad.p
Serial.p
ShutDown.p

Signal.p
Slots.p
Sound.p
Start.p
Strings.p
SysEqu.p
TextEdit.p
Timer.p
ToolIntf.p
ToolUtils.p
Traps.p
Types.p
Video.p
VideoIntf.p
Windows.p

MPW C files

Distribution disk MPW C1:

MPW C1:Libraries:CLibraries:

CInterface.o
CLib881.o
Complex.o
Complex881.o
CRuntime.o
CSANELib.o
CSANELib881.o
Math.o
Math881.o
StdCLib.o

MPW C1:Tools:

C

Distribution disk MPW C2:

MPW C2:Examples:CExamples:

Count.c
Count.r
EditCDEV.c
EditCDev.make
EditCdev.r
FStubs.c
Instructions
MakeFile
Memory.c
Memory.r
Sample.c
Sample.h
Sample.make
Sample.r
SillyBalls.c
SillyBalls.make
TESample.c
TESample.h
TESample.make
TESample.r
TESampleGlue.a
TESampleGlue.a.o
TestPerf.c
TubeTest.c
TubeTest.make
TubeTest.r

MPW C2:Interfaces:CIncludes:

AppleTalk.h
Assert.h
Complex.h
Controls.h
CType.h
CursorCtl.h
Desk.h
DeskBus.h
Devices.h
Dialogs.h
DisAsmLookUp.h
DiskInit.h

Disks.h
ErrMgr.h
ErrNo.h
Errors.h
Events.h
FCntl.h
Files.h
FixMath.h
Float.h
Fonts.h
Graf3D.h
HyperXCmd.h
IOCtl.h
Limits.h
Lists.h
Locale.h
Math.h
Memory.h
Menus.h
Notification.h
OSEvents.h
OSUtils.h
Packages.h
Palette.h
Palettes.h
Perf.h
Picker.h
Printing.h
PrintTraps.h
Quickdraw.h
Resources.h
Retrace.h
ROMDefs.h
SANE.h
Scrap.h
Script.h
SCSI.h
SegLoad.h
Serial.h
SetJmp.h
ShutDown.h
Signal.h
Slots.h
Sound.h
Start.h
StdArg.h

StdDef.h
StdIO.h
StdLib.h
String.h
Strings.h
SysEqu.h
TextEdit.h
Time.h
Timer.h
ToolUtils.h
Traps.h
Types.h
Values.h
Video.h
Windows.h

Hard disk configuration

HardDisk:MPW:

:Examples:
:Interfaces:
:Libraries:
':ROM Maps:'
:Scripts:
:Tools:
'MPW Shell'
MPW.Help
Quit
Resume
Startup
Suspend
SysErrs.Err
UserStartup
Worksheet

HardDisk:MPW:Examples:

:AExamples:
:CExamples:
:Examples:
:HyperXExamples:
:PExamples:
':Projector Examples:'

HardDisk:MPW:Examples:AExamples:

Count.a
Count.r
FStubs.a
Instructions
MakeFile
Memory.a
Sample.a
Sample.h
Sample.incl.a
Sample.make
Sample.r
SampleMisc.a

HardDisk:MPW:Examples:CExamples:

Count.c
Count.r
EditCDEV.c
EditCDev.make
EditCdev.r
FStubs.c
Instructions
MakeFile
Memory.c
Memory.r
Sample.c
Sample.h
Sample.make
Sample.r
SillyBalls.c
SillyBalls.make
TESample.c
TESample.h

TESample.make
TESample.r
TESampleGlue.a
TESampleGlue.a.o
TestPerf.c
TubeTest.c
TubeTest.make
TubeTest.r

HardDisk:MPW:Examples:Examples:

AddMenus
CheckInActive
CheckOutActive
DerezPict
Instructions
Lookup
'Startup, etc.'
State
'Unix Aliases'

HardDisk:MPW:Examples:HyperXExamples:

HardDisk:MPW:Examples:PExamples:

EditCdev.make
EditCdev.p
EditCdev.r
FStubs.a
Instructions
MakeFile
Memory.p
Memory.r
ResEqual.p
ResEqual.r
Sample.h
Sample.make
Sample.p
Sample.r
SillyBalls.make
SillyBalls.p
TESample.h
TESample.make
TESample.p
TESample.r
TESampleGlue.a

TESampleGlue.a.o
TestPerf.p
TubeTest.make
TubeTest.p
TubeTest.r

HardDisk:MPW:Examples:Projector Examples:

:Sample:

HardDisk:MPW:Examples:Projector Examples:Sample:

:Commands:
:Utilities:
ProjectorDB

HardDisk:MPW:Examples:Projector Examples:Sample:Commands:

ProjectorDB

HardDisk:MPW:Examples:Projector Examples:Sample:Utilities:

ProjectorDB

HardDisk:MPW:Interfaces:

:AIncludes:
:AStructMacs:
:CIncludes:
:PInterfaces:
:RIncludes:

HardDisk:MPW:Interfaces:AIncludes:

ApplDeskBus.a
ATalkEqu.a
FixMath.a
FSEqu.a
FSPrivate.a
Graf3DEqu.a
HardwareEqu.a
HyperXCmd.a
IntEnv.a
ObjMacros.a
PackMacs.a
PaletteEqu.a
PickerEqu.a
PrEqu.a

PrintCallsEqu.a
PrintTrapsEqu.a
Private.a
PrPrivate.a
QuickEqu.a
ROMEqu.a
SANEMacs.a
SANEMacs881.a
ScriptEqu.a
SCSIEqu.a
ShutDownEqu.a
Signal.a
SlotEqu.a
SonyEqu.a
Sound.a
SysEqu.a
SysErr.a
TimeEqu.a
ToolEqu.a
Traps.a
VideoEqu.a

HardDisk:MPW:Interfaces:AStructMacs:

FlowCtlMacs.a
ProgStrucMacs.a
Sample.a
Sample.r

HardDisk:MPW:Interfaces:CIncludes:

AppleTalk.h
Assert.h
Complex.h
Controls.h
CType.h
CursorCtl.h
Desk.h
DeskBus.h
Devices.h
Dialogs.h
DisAsmLookUp.h
DiskInit.h
Disks.h
ErrMgr.h
ErrNo.h

Errors.h
Events.h
FCntl.h
Files.h
FixMath.h
Float.h
Fonts.h
Graf3D.h
HyperXCmd.h
IOCtl.h
Limits.h
Lists.h
Locale.h
Math.h
Memory.h
Menus.h
Notification.h
OSEvents.h
OSUtils.h
Packages.h
Palette.h
Palettes.h
Perf.h
Picker.h
Printing.h
PrintTraps.h
Quickdraw.h
Resources.h
Retrace.h
ROMDefs.h
SANE.h
Scrap.h
Script.h
SCSI.h
SegLoad.h
Serial.h
SetJmp.h
ShutDown.h
Signal.h
Slots.h
Sound.h
Start.h
StdArg.h
StdDef.h
StdIO.h
StdLib.h

String.h
Strings.h
SysEqu.h
TextEdit.h
Time.h
Timer.h
ToolUtils.h
Traps.h
Types.h
Values.h
Video.h
Windows.h

HardDisk:MPW:Interfaces:PInterfaces:

AppleTalk.p
Controls.p
CursorCtl.p
Desk.p
DeskBus.p
Devices.p
Dialogs.p
DisAsmLookUp.p
DiskInit.p
Disks.p
ErrMgr.p
Errors.p
Events.p
Files.p
FixMath.p
Fonts.p
Graf3D.p
HyperXCmd.p
IntEnv.p
Lists.p
MacPrint.p
Memory.p
MemTypes.p
Menus.p
Notification.p
ObjIntf.p
OSEvents.p
OSIntf.p
OSUtils.p
Packages.p

PackIntf.p
PaletteMgr.p
Palettes.p
PasLibIntf.p
Perf.p
Picker.p
PickerIntf.p
Printing.p
PrintTraps.p
Quickdraw.p
Resources.p
Retrace.p
ROMDefs.p
SANE.p
Scrap.p
Script.p
SCSI.p
SCSIIntf.p
SegLoad.p
Serial.p
ShutDown.p
Signal.p
Slots.p
Sound.p
Start.p
Strings.p
SysEqu.p
TextEdit.p
Timer.p
ToolIntf.p
ToolUtils.p
Traps.p
Types.p
Video.p
VideoIntf.p
Windows.p

HardDisk:MPW:Interfaces:RIncludes:

Cmdo.r
MPWTypes.r
Pict.r
SysTypes.r
Types.r

HardDisk:MPW:Libraries:

:CLibraries:
:Libraries:
:PLibraries:

HardDisk:MPW:Libraries:CLibraries:

CInterface.o
CLib881.o
Complex.o
Complex881.o
CRuntime.o
CSANELib.o
CSANELib881.o
Math.o
Math881.o
StdCLib.o

HardDisk:MPW:Libraries:Libraries:

DRVRRuntime.o
HyperXLib.o
Interface.o
ObjLib.o
PerformLib.o
Runtime.o
SERD
Stubs.o
ToolLibs.o

HardDisk:MPW:Libraries:PLibraries:

PasLib.o
SANELib.o
SANELib881.o

HardDisk:MPW:ROM Maps:

MacIIROM.map
MacPlusROM.map
MacSEROM.map

HardDisk:MPW:Scripts:

BuildCommands
BuildMenu
BuildProgram
CCvt
CompareFiles
CompareRevisions
CPlus
CreateMake
DirectoryMenu
DoIt
Line
MergeBranch
OrphanFiles
SetDirectory
TransferCKID
UserVariables

HardDisk:MPW:Tools:

AboutBox
Asm
Backup
C
Canon
Canon.Dict
CCvtMxL.dict
CCvtUMx.dict
CFront
Choose
Commando
Compare
Count
DeRez
DumpCode
DumpFile
DumpObj
Entab
FileDiv
GetErrorText
GetFileName
GetListItem
Lib
Link
Make

MakeErrorFile
MatchIt
Pascal
PasMat
PasRef
PerformReport
Print
ProcNames
ResEqual
Rez
RezDet
Search
SetPrivilege
SetVersion
Sort
Translate
WhereIs

Appendix B Summary of Selections and Regular Expressions

THIS APPENDIX FORMALLY DEFINES THE SYNTAX OF SELECTIONS AND REGULAR EXPRESSIONS as used in the MPW Shell command language. It also lists the Option-key characters used in selections and regular expressions. For examples of their use, see Chapter 6. ■

Contents

Selections

Selections are passed as arguments to the editing commands. They're defined in Table B-1.

■ **Table B-1**   Selections

selection (specifies a selection or insertion point)

§	Current selection
name	Identifies marked text
number	Line number
! *number*	*number* lines after the end of the current selection
¡ *number*	*number* lines before the start of the current selection
position	Position (defined below)
pattern	Pattern (defined below)
(*selection*)	Selection grouping
selection : *selection*	Both selections and everything in between

position (specifies an insertion point)

•	Position before the first character in the file
∞	Position after the last character in the file
Δ *selection*	Position before the first character of *selection*
selection Δ	Position after the last character of *selection*
selection ! *number*	Position *number* characters after the end of *selection*
selection ¡ *number*	Position *number* characters before the beginning of *selection*

pattern (specifies characters to be matched)

/ *entireRegularExpr* /	Regular expression—search forward (see Table B-2)
\ *entireRegularExpr* \	Regular expression—search backward

This is the precedence of the selection operators, from highest to lowest:

/ and \
( )
Δ
! and ¡
:

Regular expressions

Regular expressions are used for pattern matching within /.../ and \...\. (See "pattern" in Table B-1.) Regular expressions are defined in Table B-2.

■ **Table B-2**   Regular expressions

entireRegularExpr

• *regularExpr*	Regular expression at beginning of line
regularExpr ∞	Regular expression at end of line
regularExpr	Regular expression

regularExpr

simpleExpr	Untagged regular expression
taggedExpr	Tagged regular expression
literal	Quoted string literal
regularExpr₁ regularExpr₂	*regularExpr₁* followed by *regularExpr₂*

simpleExpr

(*regularExpr*)	Regular expression grouping
characterExpr	Single-character regular expression
simpleExpr *	Regular expression zero or more times
simpleExpr +	Regular expression one or more times
simpleExpr «*number*»	Regular expression *number* times
simpleExpr «*number*,»	Regular expression at least *number* times
simpleExpr « n_1 , n_2 »	Regular expression at least n_1 times and at most n_2 times

taggedExpr

(*regularExpr*)®*digit*	The string matched by the *regularExpr* can be referred to as ®*digit* (where 0 ≤*digit*≥ 9)

literal

'*string*'	Each character in *string* is taken literally
"*string*"	Each character in *string* is taken literally, except for ∂, { }, and `...` substitutions

(Continued)

characterExpr	
character	Character (unless it's listed as special in the following table)
∂character	∂ defeats special meaning of following character
?	Any character except Return
≈	Any string not containing a Return, including the null string (this is the same as ?*)
[*characterList*]	Any character in the list
[¬ *characterList*]	Any character not in the list

characterList	
]	"]" first in list represents itself
–	"–" first in list represents itself
character	Character
characterList character	List of characters
character₁ – character₂	Character range from *character₁* to *character₂* inclusive

◆ *Note:* The regular expression operators

   ?   ≈   [...]   *   +   «...»

are also used in filename generation.

The following characters have special meanings:

∂	Always special, except within '...'
? ≈ * + [« () ' "	Special everywhere except within [...], '...', and "..."
®	Special only after a right parenthesis character,)
•	Special as first character of entire regular expression
∞	Special as last character of entire regular expression
/ \	Special if used to delimit regular expression
{ }	Special everywhere except within '...'
¬	Special immediately following left bracket [
–	Special within brackets except immediately following left bracket [

The operators are listed below beginning with those with the highest precedence.

( )

?   ≈   *   +   [ ]   « »   ®

concatenation

•   ∞

Option-key characters

The following Option-key characters are used in selections and regular expressions.

◆ *Note:* Option-key characters are not case-sensitive. Although upper case letters are shown in the text of this reference for readability (the number 1 and lower case L look the same), you can use lower case for all Option-letter characters.

Character	Key	Meaning
§	Option-6	Current selection character
∂	Option-D	Escape character
≈	Option-X	Any string
•	Option-8	Beginning of line or file
∞	Option-5	End of line or file
¡	Option-1	Minus number of lines or spaces
∆	Option-J	Position
®	Option-R	Tag operator
«	Option-\	Encloses number of repetitions
»	Option-Shift-\	Encloses number of repetitions
¬	Option-L	Character list modifier

Appendix C Special Operators

HERE IS A BRIEF SUMMARY OF THE SPECIAL OPERATORS USED IN MPW 3.0. For characters that are part of the extended character set, Option-key combinations are also given. For details on the action of these operators, see Chapters 5 and 6. See Appendix B for a summary of selections and regular expressions. ∎

■ **Table C-1**   MPW operators

Operator	Meaning

Shell character

space	Separates words
tab	Separates words
return	Separates commands
;	Separates commands
|	Pipe—separates commands, piping output to input
&&	"And"—separates commands, executing second if first succeeds
||	"Or"—separates commands, executing second if first fails
(*commands*)	Group commands
# *comment*	Ignore *comment*
∂*char*	Escape—literalizes *char*; ∂n, ∂t, and ∂f are special (∂ is Option-D)
'*chars*'	"Hard quotation marks"—literalize *chars*
"*chars*"	"Soft quotation marks"—literalize *chars* except for {...} (variable substitution), `...` (command substitution), and ∂ (escape)
/*chars*/	Regular expression quotes—literalize /*chars*/ except for {...}, `...`, and ∂
...	Ellipsis (Option-semicolon; not three periods) following a command invokes Commando
chars\\	Regular expression quotation marks—literalize *chars*\\ except for {...}, `...`, and ∂
{*variable*}	Substitute *variable*
`command`	Substitute output of *command*

I/O redirection

Note: Filename is created if it does not exist.

< *filename*	Standard input is taken from filename
> *filename*	Redirect standard output, replacing contents of *filename*
>> *filename*	Redirect standard output, appending to *filename*
≥ *filename*	Redirect diagnostics, replacing contents of *filename* (≥ is Option->)
≥≥ *filename*	Redirect diagnostics, appending to *filename* (≥ is Option->)
Σ *filename*	Redirect both standard output and diagnostics, replacing contents of *filename* (Σ is Option-W)
ΣΣ *filename*	Redirect both standard output and diagnostics appending to *filename* (Σ is Option-W)

(Continued)

Operator			Meaning

Shell numbers

$ [0–9 a–f]+			Hexadecimal number
0x [0–9 a–f]+			Hexadecimal number
0 [0–7]+			Octal number
0b [0–1]+			Binary number

Shell operators (by precedence)

(*expr*)			Expression grouping
–			(unary) arithmetic negation
~			(unary) bitwise negation
!	NOT	¬	(unary) logical negation (¬ is Option-L)
*			Multiplication
÷	DIV		Division (÷ is Option-/)
%	MOD		Modulus
+			Addition
–			Subtraction
<<			Shift left
>>			Shift right (logical)
<			Less than
<=	≤		Less than or equal (≤ is Option-<)
>			Greater than
>=	≥		Greater than or equal (≥ is Option->)
==			Equal
!=	<>	≠	Not equal (≠ is Option-=)
=~			Equal to a pattern
!~			Not equal to a pattern
&			Bitwise AND
∧			Bitwise XOR
I			Bitwise OR
&&	AND		Logical AND
I I	OR		Logical OR

Appendix D Resource Description Syntax

THIS APPENDIX DEFINES THE FORM OF RESOURCE DESCRIPTION FILES used by the
MPW 3.0 resource compiler (Rez) and decompiler (DeRez). See Chapter 11 for
information on how to use these tools. Each tool is defined in detail along with
examples in Part II. ■

Contents

Syntax notation

The following syntax notation is used in this appendix:

`terminal`	Must be entered as shown
nonterminal	May be replaced by anything matching its definition
`A	B	C`	Either A or B or C (vertical stacking also indicates an either/or choice)
{...}?	Enclosed element is optional, but may not be repeated
{...}+	Enclosed element may be repeated one or more times (not optional)
{...}*	Enclosed element may be repeated zero or more times
{...}*n*	Enclosed element must be repeated *n* times

If one of the syntax elements must be included literally, it is shown enclosed in single quotation marks; for example,

```
{ `{' data-string `}' }?
```

indicates that a *data-string* is optional, and must be enclosed in braces, if included. Otherwise, all punctuation (; , ' " $ =) must be entered as shown.

Note that the ellipsis (three closely spaced periods) within braces signifies only some unspecified element on which an operation is to be performed. An actual ellipsis in a command line (Option-semicolon) would invoke a command's Commando dialogs.

Note that the semicolon is a statement terminator; every statement must be terminated by a semicolon. In a resource type definition, semicolons can be liberally sprinkled without ill effect. In a resource specification (where the actual resource data is initialized), commas are used everywhere to separate items, including array elements.

The nonterminal symbols used are fully defined under "Syntax" at the end of this appendix.

Structure of a resource description file

The MPW resource compiler input file consists of any number of statements, where a statement may be any of the following:

include	Include resources from another file.
read	Read the data fork of a file and include it as a resource.
data	Specify raw data.
type	Declare resource type descriptions for subsequent `resource` statements.
resource	Specify data for a resource type declared in a previous `type` statement.
change	Change the `type`, ID, name, or attributes of existing resources.
delete	Delete existing resources.

Include—include resources from another file

```
include file { include-selector }? ;
```

include-selector ::=	*resource-type* { ' (' *ID-specifier* ')' }? not *resource-type* *resource-type1* as *resource-type2* *resource-type1* ' (' *ID-specifier* ')' as *resource-type2* ' (' *resource-specifier* ')'
file ::=	*string*
ID-specifier ::=	*ID-range* *resource-name*
ID-range ::=	*ID* { : *ID* }?
resource-specifier ::=	*resource-ID* { , *resource-name* }? { *resource-attributes* }?
resource-ID ::=	*word-expression*
resource-name ::=	*string*
resource-attributes ::=	{ *resource-literal-attributes* } *	*resource-numeric-attributes*
resource-numeric-attributes ::=	, *byte-expression*
resource-literal-attributes ::=	{ , sysheap	, appheap }? { , purgeable	, nonpurgeable }? { , locked	, unlocked }? { , preload	, nonpreload }?

Read—read data as a resource

```
read resource-type ' (' resource-specifier ')' file ;
```

Data—specify raw data

```
data resource-type ' (' resource-specifier ')' '{' data-string { ; }? '}' ;
```

Type—declare resource type

type *resource-type* { ' ('*ID-range* ') ' }? '{' { {label ':'} * *type-statement* ;} * '}' ;

resource-type ::=	*long-expression*
type-statement ::=	*data-type*
	fill-type
	alignment
	switch-type
	array-type
label ::=	*identifier*

Data-type

data-type ::= *data-type-specifier* { *symbolic-declaration* | = *declaration-constant* }?

data-type-specifier ::=	`char`
	`string` { '[' *length* ']' }?
	`pstring` { '[' *length* ']' }?
	`cstring` { '[' *length* ']' }?
	`wstring` { '[' *length* ']' }?
	numeric-type-specifier
	`point`
	`rect`
length ::=	*expression*
numeric-type-specifier ::=	`boolean`
	{ `unsigned` }? { *radix* }? *numeric-type*
radix ::=	`binary`
	`octal`
	`decimal`
	`hex`
	`literal`
numeric-type ::=	`byte`
	`integer`
	`longint`
	`bitstring` '[' *length* ']'
symbolic-declaration ::=	*range-block* { , *range-block* }*

declaration-constant ::=	*expression*
	point-constant
	rect-constant
	string

Fill-type

fill-type ::=	`fill` *fill-size* {'[' *expression* ']'}?
fill-size ::=	`bit` | `nibble` | `byte` | `word` | `long`

Alignment

alignment ::=	`align` *align-size*
align-size ::=	`nibble` | `byte` | `word` | `long`

Switch-type

switch-type ::=	`switch` '{' *switch-body* '}'
switch-body ::=	{ `case` *case-name* : *case-body* }+
case-name ::=	*identifier*
case-body ::=	{ *type-statement* ; }* *key-constant-statement* ; { *type-statement* ; }*
key-constant-statement ::=	`key` *data-type-specifier* = *declaration-constant*

Array-type

array-type ::=	{`wide`}? `array` { *array-specifier* }? *type-body*
array-specifier ::=	*array-name*
	'[' *expression* ']'
array-name ::=	*identifier*
type-body ::=	'{' { *type-statement* ; } * '}'

Resource—specify resource data

resource *resource-type* '(' *resource-specifier* ')' *data-body* ;

data-body ::=	'{' { *data-statement* { , *data-statement* }* }? '}'
data-statement ::=	*expression*
	point-constant
	rect-constant
	string
	identifier
	switch-data
	array-data
switch-data ::=	*case-name data-body*
array-data ::=	'{' { *array-element* { , *array-element* }* }? '}'
	'{' { *array element* ; }* '}'
array-element ::=	{ *data-statement* { , *data-statement* }* }?

Change—change resource vital information

change *resource-type* { '(' *ID-specifier* ')' }? to *resource-type2* '(' *resource-specifier* ')' ;

Delete—delete resource(s)

delete *resource-type* { '(' *ID-specifier* ')' }? ;

Labels

Labels support some of the more complicated resources such as 'NFNT' and color QuickDraw resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels.

Syntax

```
label     ::=   character {alphanum}* ':'
character ::=   '_' | A | B | C ...
number    ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum  ::=   character | number
```

Preprocessor directives

These preprocessor directives are available:

```
#define  identifier{ define-string }?
#undef identifier
#if  preprocessor-expr
#elif  preprocessor-expr
#else
#endif
#ifdef  identifier
#ifndef  identifier
#printf ( string{ ,[ expression | string ]}* )
```

Preprocessor-expr is the same as *expression* with the following additional expressions:

```
defined '('identifier')'
defined identifier
```

Syntax

This section defines the nonterminal symbols used in the previous sections.

Identifiers

- An *identifier* may consist of letters (A–Z, a–z), digits (0–9), or the underscore character (_).

- Identifiers may not start with a digit; otherwise any mix of letters, digits, and underscores is acceptable.

- Identifiers are not case sensitive.

- An identifier may be of any length.

Token delimiters

token-delimiter ::=   { *space* | *tab* | *newline* | *comment* }+

comment ::=   '/*' { *printing-character* }* '*/'
                                    // { *printing-character* }* *newline*

Compound types

point-constant ::=   '{' *expression* , *expression* '}'
rect-constant ::=   '{' *expression* , *expression* , *expression* , *expression* '}'

Expressions

bit-expression ::=   *expression*
byte-expression ::=   *expression*
word-expression ::=   *expression*
long-expression ::=   *expression*

expression ::=   *integer-constant*
                                    literal-constant
                                    numeric-variable
                                    system-function
                                    expression
                                    label
                                    – *expression*
                                    ~ *expression*
                                    ! *expression*
                                    '(' *expression* ')'
                                    expression   >>   *expression*
                                    expression   <<   *expression*
                                    expression   ^   *expression*
                                    expression   '| |' *expression*
                                    expression   &&   *expression*
                                    expression   '|'   *expression*
                                    expression   &   *expression*
                                    expression   !=   *expression*
                                    expression   ==   *expression*
                                    expression   >=   *expression*
                                    expression   <=   *expression*

.

expression	>	*expression*
expression	<	*expression*
expression	–	*expression*
expression	+	*expression*
expression	*	*expression*
expression	/	*expression*
expression	%	*expression*

system-function ::= $$countof '(' *array-name* ')'

$$packedsize '(' *StartOffSet, RowBytes, RowCount* ')'

Numbers

integer-constant ::= *decimal-constant*
octal-constant
binary-constant
hexadecimal-constant

decimal-constant ::= *nonzero-digit* { *digit* }*
octal-constant ::= 0 { *octal-digit* }*
hexadecimal-constant ::= *hex-marker* { *hex-digit* }+
binary-constant ::= *binary-marker* { *binary-digit* }+

decimal-marker ::= 0d | 0D
hex-marker ::= 0x | 0X | $
binary-marker ::= 0b | 0B

octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
A | B | C | D | E | F |
a | b | c | d | e | f

binary-digit ::= 0 | 1

literal-constant ::= ' { *character* }* '

Variables and functions

string-variable ::=

$$Version

$$Date

$$Time

$$Name

$$Shell '(" *Shell-variable-name* ")'

$$Resource '('*file, resource-id, resourceName-or-ID*)'

$$Format (*string* {, [*expression* | *string*]} *)

    resourceName-or-ID ::= *resource-id*

                   resource-name

numeric-variable ::=

$$Hour

$$Minute

$$Second

$$Year

$$Month

$$Day

$$Weekday

$$Type

$$ID

$$Attributes

$$ResourceSize

$$BitField (*expression, expression, expression*)

$$Byte (*expression*)

$$Long (*expression*)

$$PackedSize (*expression, expression, expression*)

$$Word (*expression*)

Strings

string ::=	*simple-string*
	hex-string
	string-variable
	string string
simple-string ::=	" { *character* }* "
hex-string ::=	$" { *hex-digit hex-digit* }* "
character ::=	*printing-character*	*escape-character*
escape-character ::=	\ *escape-code*
escape-code ::=	*character-escape-code*	*numeric-escape-code*
character-escape-code ::=	n	t	b	r	f	v	?	\	'	"
numeric-escape-code ::=	{ *octal-digit* }3
	decimal-marker{ *decimal-digit* }3
	hex-marker{ *hex-digit* }2
	binary-marker{ *binary-digit* }8

Appendix E **File Types, Creators, and Suffixes**

Contents

File types and creators

Table E-1 lists MPW file types and creators.

♦ Note: The file type `'OBJ '` actually contains a space before the closing single quotation mark. Likewise, the creator `'MPS '` has a space before its closing quotation mark.

■ **Table E-1**   File types and creators

File	Type	Creator	
MPW Shell	`'APPL'`	`'MPS '`	(MPS*space*)
Tools	`'MPST'`	`'MPS '`	
Text files	`'TEXT'`	`'MPS '`	
Object files	`'OBJ '`	`'MPS '`	
Assembler load/dump	`'DMPA'`	`'MPS '`	
C load/dump	`'DMPC'`	`'MPS '`	
Pascal load/dump	`'DMPP'`	`'MPS '`	
ProjectorDB	`'MPSP'`	`'MPS '`	
SADE	`'APPL'`	`'sade'`	
SADE text files	`'TEXT'`	`'sade'`	
SADE symbol files	`'MPSY'`	`'sade'`	
SADE symbol files	`.sym`	`'sade'`	

File suffixes

The following sections define file suffix conventions.

Text files

name.a	Assembly-language source file
name.a.lst	Assembler listing file
name.c	C or C++ source file
name.cp	C++ source file
name.h	C header file
name.map	Linker map
name.p	Pascal source file
name.r	Resource description file (resource compiler (Rez) input)

Text files are identified by their file type ('TEXT') rather than by a special suffix. Several applications (including MacWrite, MDS Edit, and the MPW Shell) can create and edit files of type 'TEXT'. The creator 'MPS' indicates to the Finder that the MPW Shell is the application to launch when a text file is opened.

Object files

name.a.o	Object file created by the assembler
name.p.o	Object file created by the MPW Pascal Compiler
name.c.o	Object file created by the MPW C Compiler
name.o	Object file (library) created by Lib; object files shipped with MPW

Compilers add the suffix ".o" to the source file name to construct the object file name. The language suffix is left in the name in order to prevent name conflicts for programs whose components are written in several languages. (For example, a program might have source files MacGismo.a and MacGismo.c and object files MacGismo.a.o and MacGismo.c.o.)

Data files

ProjectorDB	Database file created by Projector
name.SYM	Symbolic information file created by the linker

The linker adds the suffex ".SYM" to the output filename in response to the -sym option.

Appendix F   Tools Libraries

THE MPW TOOLS LIBRARY, FOUND IN TOOLLIBS.O, INCLUDES PROCEDURES AND HEADER FILES to

■   control the MPW rotating beach ball cursor

■   retrieve the text of Macintosh Operating System error messages from the MPW error message file

■   dissassemble MC68xxx machine code

MPW Assembly language programmers can use either MPW Pascal or MPW C calls to all of these routines. Therefore, only the Pascal and C calling conventions are shown here. (However, you will find special notes for Assembler users.)   ■

Contents

Animated cursor control routines

Five procedures in the MPW Tools Library let you control the appearance and action of the MPW cursor. The rotating beach ball cursor says "I am currently processing." These routines all use Pascal calling conventions and Pascal-style strings.

◆ *Note:* Spinning the cursor allows your tool to operate in the background under MultiFinder.

Cursor control routines—MPW Pascal

To access the cursor control unit in MPW Pascal, do the following:

■ Include these statements in your source text:

```
USES {$U MemTypes.p} MemTypes,
     {$U CursorCtl.p} CursorCtl;
```

The USES clause and the $U compiler directive are described in the *MPW 3.0 Pascal Reference.*

■ Link your compilation with the file ToolLibs.o.

Cursor control routines—MPW C

The MPW C header file CursorCtl.h provides interfaces to procedures in the MPW Tools Library that let you control the appearance and action of the cursor. Link this file with the file ToolLibs.o.

To access the cursor control unit in MPW C, include this statement in your source text:

```
#include <CursorCtl.h>
```

The InitCursorCtl procedure

The `InitCursorCtl` procedure initializes the `CursorCtl` unit. Call this procedure once prior to calling the `RotateCursor` or `SpinCursor` procedures described later in this appendix. Note that `InitCursorCtl` doesn't need to be called if you use only `Hide_Cursor` and `Show_Cursor`.

If the parameter *NewCursors* is `NIL`, `InitCursorCtl` loads in the 'acur' resource and the 'CURS' resources specified by the 'acur' resource ID. If any of the resources cannot be loaded, the cursor will not be changed. The 'acur' resource is assumed to be either in the currently running tool or application or the MPW 3.0 Shell for a tool or in the System file. The 'acur' resource ID must be 0 for a tool or application, 1 for the Shell, and 2 for the System file (assuming that cursors are in the System file).

If *NewCursors* is not `NIL`, it is assumed to be a handle to an 'acur'-formatted resource designated by the caller and uses it instead of executing the `GetResource` procedure on 'acur'.

♦ *Note:* If you call `RotateCursor` or `SpinCursor` without first calling `InitCursorCtl`, then `RotateCursor` and `SpinCursor` do the work of `InitCursor` the first time you make the call. However, it is preferable to call `InitCursorCtl` first because of one possible disadvantage: The resource memory allocated may cause fragmentation to occur in the application. Calling `InitCursorCtl` has the advantage of allocating memory at a time you specify.

`CursorCtl` declares `acurHandle` as a handle to 'acur' resources of type `RECORD` as follows:

```
TYPE
    acurHandle = ^acurPtr;          {Handles to 'acur' resources}
    acurPtr    = ^acur;             {Pointers to 'acur' resources}

    acur      =       RECORD        {Layout of an 'acur' resource}
              N:      integer;      {Number of cursors ("frames of film")}
              Index:  integer       {Next frame to show <for internal use>}
              Frame1: integer       {'CURS' resource ID - frame #1}
              fill1:  integer       {<for internal use>}
              Frame2: integer       {'CURS' resource ID - frame #2}
              fill2:  integer       {<for internal use>}
{- - -- - - - -- - - - -- - - - -- - - -}
              FrameN: integer       {'CURS' resource ID - frame #2}
              fillN:  integer       {{<for internal use>}
    END;
```

See "The RotateCursor Procedure" for a description of how the `acur` frames are used to animate the cursor.

▲ Warning   InitCursorCtl modifies the `acur` resource in memory. Specifically, it changes each FrameN/fillN integer pair to a handle to the corresponding `curs` resource also in memory. Thus if NewCursors is not NULL when InitCursorCtl is called, you must guarantee that NewCursors always points to a "fresh" copy of an `acur` resource. This need concern you only if you want to repeatedly use multiple `acur` resources during execution of your tools. ▲

MPW Pascal

```
InitCursorCtl(NewCursors: UNIV acurHandle);
```

MPW C

```
pascal void InitCursorCtl(acurHandle newCursors);
```

The Show_Cursor procedure

The Show_Cursor procedure increments the cursor level (which may have been decremented by Hide_Cursor). If the level is zero, it displays the cursor. The cursor level never increments above zero. The parameter *CursorKind* lets you select the form of the cursor:

■ **Table F-1**   Cursor kinds

Value	Cursor
0	Hidden cursor
1	I-beam
2	Cross
3	Plus sign
4	Watch
5	Arrow

Except for HIDDEN_CURSOR, a Macintosh SetCursor is done for the specified cursor prior to doing a ShowCursor. HIDDEN_CURSOR simply causes a ShowCursor call.

◆ *Note:* IntiGraf( ) must be called before any calls to ShowCursor
(ARROW_CURSOR) because the arrow cursor is one of the QuickDraw globals set up by
IntiGraf( ).

MPW Pascal

```
Show_Cursor(CursorKind: Cursors);
```

CursorCtl declares the type Cursors as follows:

```
TYPE
    Cursors = (HIDDEN_CURSOR, I_BEAM_CURSOR, CROSS_CURSOR,
               PLUS_CURSOR, WATCH_CURSOR, ARROW_CURSOR);
```

MPW C

```
pascal void Show_Cursor(Cursors cursorKind);
```

The Hide_Cursor procedure

The Hide_Cursor procedure calls the Macintosh HideCursor routine. (Thus the Macintosh cursor level is decremented by 1 when this routine is called.) If the cursor was visible, it is then hidden. For further information, see the chapter "QuickDraw" of *Inside Macintosh.*

MPW Pascal

```
Hide_Cursor;
```

MPW C

```
pascal void  Hide_Cursor(void)
```

The RotateCursor procedure

The `RotateCursor` procedure rotates the beach ball cursor (or animates whichever sequence of cursors has been set by the user in an `'acur'` resource and loaded with `InitCursorCtl`) and rotates it one-quarter turn (that is, advances to the next `'acur'` resource frame) whenever the value of *Counter* is a multiple of 32. To use `RotateCursor`, your program must set up and increment (or decrement) a suitable counter. If the value of *Counter* is positive, the cursor rotates clockwise (that is, sequencing is forward through the `'acur'` cursor frames); if it is negative, it rotates counterclockwise (that is, sequencing is backward circularly through the `'acur'` resource frames).

◆ *Note:* `RotateCursor` invokes a Macintosh `SetCursor` call for the proper cursor picture. It assumes that the cursor is visible as the result of a prior `show_Cursor` call.

MPW Pascal

```
RotateCursor(Counter: longint);
```

MPW C

```
pascal void RotateCursor(long counter);
```

The SpinCursor procedure

The `SpinCursor` procedure performs the same actions as `RotateCursor`, but maintains its own internal counter rather than passing a counter. It is provided for those who do not have a convenient counter handy but still want to use the spinning beach ball cursor or any sequence of cursors specified by `InitCursorCtl`. Your program specifies the *Increment* to be counted (either positive or negative), and `SpinCursor` adds it to its counter. A positive increment spins the cursor clockwise (that is, sequencing is forward through the `'acur'` cursor frames); a negative increment spins it counterclockwise (that is, sequencing is backward circularly through the `'acur'` resource frames). An *Increment* value of zero resets the counter to zero.

◆ *Note:* It is the sign of the increment, not the sign of the accumulated value of the `SpinCursor` counter, that determines the cursor's direction of spin.

MPW Pascal

```
SpinCursor(Increment: integer);
```

MPW C

```
pascal void SpinCursor(short increment);
```

Error Message File manager

Four procedures in the MPW Tools Library let you retrieve the text of error messages in the Macintosh Operating System error message file or in an error file private to a tool (created with the MakeErrorFile tool).

Error Manager—MPW Pascal

To use the error message file manager in MPW Pascal, do the following:

■ Include the statement

```
USES {$U MemTypes.p} MemTypes, {$U ErrMgr.p} ErrMgr;
```
in your source text. The USES clause and the $U compiler directive are described in the *MPW 3.0 Pascal Reference*.

■ Link your compilation with the file ToolLibs.o.

Error Manager—MPW C

Use the header file ErrMgr.h which includes the file Types.h. Link this file with ToolLibs.o.

```
#include <ErrMgr.h>
```

The InitErrMgr procedure

The `InitErrMgr` procedure must be called before any of the other error message file manager procedures. To access Macintosh Operating System error messages, use the Pascal call

```
InitErrMgr('', '', false);
```

This call causes the error manager to access the file SysErr.Err in the directory {ShellDirectory} if that Shell variable is defined; otherwise, it will use file SysErr.Err.

If `InitErrMgr` is not explicitly called, then `GetSysErrText` or `GetToolErrText` will call `InitErrMgr('', '', TRUE)` the first time they are called.

If you wish to access a tool-specific error file, supply the name of the error file as the first parameter to `InitErrMgr`. If the tool is an MPW tool with the error file copied into the tool's data fork, the first parameter may be the null string and the `ErrMgr` will open the appropriate file. This occurs only if CRuntime.o or PasLib.o is linked with the program.

Set ShowToolErrNbrs to TRUE if you want all messages to begin with the error number, as in

```
<msgtxt> ([OS]Error<n>)
```

Failure by the Error Manager to find the message text always results in a message of this form (without the `<msgtxt>`). `ToolErrFileName` is used to specify the name of the tool-specific error file, and should be the null string if not used (or if the tool's data fork is to be used as the error file). Use `SysErrFileName` to specify the name of the system error file. This should normally be the null string which causes the Error Manager to look in the MPW Shell directory for "SysErrs.Err". Specifying names for the error files avoids `IntEnv` calls that look up the values of Shell variables.

MPW Pascal

```
PROCEDURE InitErrMgr(toolErrFilename: Str255;sysErrFilename:
     Str255;showToolErrNbrs: BOOLEAN);
```

MPW C

```
InitErrMgr(Str255 toolErrFilename,Str255 sysErrFilename,
     Boolean showToolErrNbrs);
```

◆ *Note:* The Assembler caller must define and export the variable _EnvP with a null value
if CRuntime.o or PasLib.o is not linked with the tool. For example, outside all
modules (procs) place the following:

```
          EXPORT       _EnvP
EnvP      DC.L         Ø
```

The GetSysErrText procedure

The `GetSysErrText` procedure fetches the message text that corresponds to the
system error number value of *MsgNbr. ErrMsg* is a pointer to a string of type `str255`, in
which the error message text will be placed. The maximum length of the message is limited
to 254 characters.

If `GetSysErrText` is successful (and if `ShowToolErrNbrs` is true on the init call), the
form of the error message returned is

error text (OS error number)

If it is unsuccessful, the form of the error message returned is

OS error number (reason message not found)

Possible reasons for unsuccessful execution of `GetSysErrText` are that the file
SysErr.Err was not found or that it contained no message text corresponding to *MsgNbr.*

◆ *Note:* If a system message filename was not specified to `InitErrMgr`, then the error
manager assumes the message file contained in the file SysErrs.Err. This file is first
accessed as {ShellDirectory}SysErrs.Err on the assumption that SysErrs.Err is kept in
the same directory as the MPW Shell. If the file cannot be opened, then the error
manager attempts to open SysErrs.Err in the System Folder.

MPW Pascal

```
PROCEDURE GetSysErrText(msgNbr: INTEGER;errMsg: StringPtr);
```

MPW C

```
void GetSysErrText(short msgNbr,char *errMsg);
```

Get error message text corresponds to the error number `msgNbr` from the system error message file ("SysErrs.Err" in {ShellDirectory}). The text of the message is returned in `errMsg`.

The GetToolErrText procedure

The `GetToolErrText` procedure fetches the message text that corresponds to the tool error message file error number `msgNbr`. (The tool error filename is specified in the `InitErrMgr` call.) The text message is returned in `errMsg`.

Inserts are indicated in error messages by specifying a "^" (Up Arrow) to indicate where the insert is to be placed. Any message to be inserted should be contained in `errInsert`. Otherwise, `errInser` should be null. The error insert is placed in the text of the error message replacing the first instance of the "^" character in the message; if no, "^" is present, the error insert is appended to the end of the text of the message following an intervening blank.

◆ *Note:* If a tool message filename was not specified to `InitErrMgr`, then the error manager assumes the message file contained in the data fork of the tool calling the error manager. This name is contained in the Shell variable {Command} and the value of that variable is used to open the error message file.

MPW Pascal

```
PROCEDURE   GetToolErrText(msgNbr: INTEGER;errInsert: Str255;errMsg:
            StringPtr);
```

MPW C

```
void GetToolErrText(short msgNbr,char *errInsert,char *errMsg);
```

The AddErrInsert procedure

The `AddErrInsert` procedure adds another insert to an error message string. This call may be used when more than one insert is needed in a message (because it contains more than one "^" character). The insert is handled in the same fashion as in the `GetToolErrText` call.

MPW Pascal

```
PROCEDURE AddErrInsert(insert: Str255;msgString: StringPtr); C;
```

MPW C

```
void AddErrInsert(unsigned char *insert,unsigned char *msgString);
```

The CloseErrMgr function

Ideally you should call `CloseErrMgr` at the end of execution to make sure all files opened by the error manager are closed. You can let normal program termination do the closing.

MPW Pascal

```
PROCEDURE CloseErrMgr; C;
```

MPW C

```
void CloseErrMgr(void);
```

Disassembler Lookup routines

The Disassembler Lookup is an interface (available in MPW Pascal and MPW C) to the Macintosh libraries. It is a Pascal routine that disassembles a sequence of bytes.

All MC68xxx family instructions are supported, including MC68881, MC68882, and MC68851 instructions. The sequence of bytes to be disassembled are pointed to by FirstByte. BytesUsed bytes starting at FirstByte are consumed by the disassembly, and the Opcode, Operand, and Comment strings returned as NULL TERMINATED Pascal strings (for easier manipulation with C). You are then free to format or use the output strings in any way appropriate to the application.

The Pascal interface file DisAsmLookUp.p is located in the PInterfaces folder. The C interface file, DisAsmLookUp.h, is located in the CInterfaces folder. A discussion of each of these interface files and a general explanation of the Disassembler follows.

DisAsmLookUp.p—MPW Pascal

```
TYPE

LookupRegs =
(_A0_,_A1_,_A2_,_A3_,_A4_,_A5_,_A6_,_A7_,_PC_,_ABS_,_TRAP_);

DisAsmStr80 = String[80];



PROCEDURE    Disassembler(DstAdjust: LONGINT;VAR BytesUsed:
INTEGER;

             FirstByte: UNIV Ptr; VAR Opcode: UNIV DisAsmStr80;
             VAR Operand: UNIV DisAsmStr80;VAR Comment: UNIV
DisAsmStr80;
             LookUpProc: UNIV Ptr);
```

DisAsmLookUp.h—MPW C

```
enum {_A0_,_A1_,_A2_,_A3_,_A4_,_A5_,_A6_,_A7_,_PC_,_ABS_,_TRAP_};
typedef unsigned char LookupRegs;

pascal void Disassembler(long DstAdjust,short *BytesUsed,Ptr FirstByte,
     char *Opcode,char *Operand,char *Comment,Ptr LookUpProc);
```

Using the Disassembler

Depending on the opcode and effective addresses (EA's) to be disassembled, the Opcode, Operand, and Comment strings contain the following information:

■ **Table F-2**   Disassembler strings

Case	Opcode	Operand	Comment
Non PC-relative EA's	op.sz	EA's	
PC-relative EA's	op.sz	EA's	; address
Toolbox traps	DC.W	$AXXX	; TB XXXX
OS traps	DC.W	$AXXX	; OS XXXX
Invalid bytes	DC.W	$XXXX	; ????
Invalid byte #immediate	DC.W	$XXXX,...	; op.sz #$??XX,EA

For valid disassembly of processor instructions, Disassembler generates the appropriate MC68xxx opcode mnemonic for the Opcode string along with a size attribute when required. The source and destination EA's are generated as the Operand along with a possible comment. Comments start with a semicolon (;). Traps use a DC.W assembler directive as the Opcode, the trap word as the Operand, and a comment indicating the trap number and whether the trap is a toolbox or OS trap. As described later in this appendix, you can generate symbolic substitutions into EA's and provide names for traps.

Invalid instructions cause the string 'DC.W' to be returned in the Opcode string. Operand is '$xxxx' (the invalid word) with a comment of '; ????'.

BytesUsed is 2. This is similar to the trap call case except for the comment.

A special case is made for immediate byte operands with a nonzero high-order byte. For example, the bytes $020011FF, when actually executed, are interpreted as
ANDI.B $FF,D0.

The processor will *ignore* the high-order byte of the immediate data! Thus, the bytes may be considered as valid. Because the Disassembler has no way of knowing the context in which it is disassembling, it returns the Opcode as 'DC.W' as in the normal invalid case. However, the Operand string shows *all* the words disassembled separated with commas, and it places the possibly valid disassembly in the Operand's comment indicating the nonzero bytes. Thus, for the example $020011FF bytes, the Opcode will be 'DC.W', the Operand will be '$0200,$11FF', and the Comment '; ANDI.B #$??FF,D0'. BytesUsed in this case would be 4.

◆ *Note:* the Operand EA's are syntactically similar to but *not compatible with* the MPW Assembler! This is because the Disassembler generates byte hex constants as "$XX" and word hex constants as "$XXXX". Negative values (such as $FF or $FFFF) produced by the Disassembler are treated as long word values by the MPW Assembler. Thus it is assumed that Disassembler output will *not* be used as MPW Assembler input. If that is the goal, you must convert strings of the form $XX or $XXXX in the Operand string to their decimal equivalent.

The routine ModifyOperand is provided in the Disassembler routine to aid with the conversion process.

Since a PC-relative comment is an address, the only address that the Disassembler knows about is the address of the code pointed to by FirstByte. Generally, that may be a buffer that has no relation to "reality," that is, the actual code loaded into the buffer. Therefore, to allow the address comment to be mapped back to some actual address, you may specify an adjustment factor, specified by DstAdjust, that is *added* to the value that normally would be placed in the comment.

The Disassembler generates operand-effective address strings as a function of the effective address mode. A special case is made for A-trap opcode strings. In places where a possible symbolic reference could be substituted for an address (or a portion of an address), the Disassembler can call a user-specified routine to do the substitution (using the LookupProc parameter described later). The following table summarizes the generated effective addresses and notes where symbolic substitutions (S) can be made:

■ **Table F-3**   Disassembler: Effective addresses

Mode	Generated Effective Address	Effective Address with Substitution
0	Dn	Dn
1	An	An
2	(An)	(An)
3	(An)+	(An)+
4	-(An)	-(An)
5	∂(An)	S(An) or just S (if An=A5, ∂≥0)
6n	∂(An,Xn.Size*Scale)	S(An,Xn.Size*Scale)
6n	(BD,An,Xn.Size*Scale)	(S,An,Xn.Size*Scale)
6n	([BD,An],Xm.Size*Scale,OD)	([S,An],Xm.Size*Scale,OD)
6n	([BD,An,Xn.Size*Scale],OD)	([S,An,Xn.Size*Scale],OD)
70	∂	S
71	∂	S
72	*±∂	S
73	*±∂(Xn.Size*Scale)	S(Xn.Size*Scale)
73	(*±∂,Xn.Size*Scale)	(S,Xn.Size*Scale)
73	([*±∂],Xm.Size*Scale,OD)	([S],Xm.Size*Scale,OD)
73	([*±∂,Xn.Size*Scale],OD)	([S,Xn.Size*Scale],OD)
74	#data	#data

For A-traps, you can substitute for the DC.W opcode string. If the substitution is made, the Disassembler will generate ,Sys and/or ,Immed flags as operands for Toolbox traps and AutoPop for OS traps when the bits in the trap word indicate these settings.

	Generated Opcode	Operand	Comment	Substituted Opcode	Operand	Comment
Toolbox	DC.W	$AXXX	; TB XXXX	S	[,Sys][,Immed]	; AXXX
OS	DC.W	$AXXX	; OS XXXX	S	[,AutoPop]	; AXXX

All displacements (∂, BD, OD) are hexadecimal values shown as a byte ($XX), word ($XXXX), or long ($XXXXXXXX) as appropriate. The *Scale is suppressed if it is 1. The Size is W or L. Note that effective address substitutions can only be made for "∂(An)", "BD,An", and "*±∂" cases.

For all the effective address modes 5, 6n, 7n, and for A-traps, a coroutine (a procedure) whose address is specified by the LookupProc parameter is called by the Disassembler (if LookupProc is not NIL) to do the substitution (or A-trap comment) with a string returned by the procedure. It is assumed that the procedure pointed to by LookupProc is a level 1 Pascal procedure declared as follows:

```
PROCEDURE Lookup( PC:UNIV Ptr;          {Addr of extension/trap word}
                  BaseReg: LookupRegs;   {Base register/lookup mode  }
                  Opnd:UNIV LongInt;     {Trap word, PC addr, disp.  }
           VAR S:     DisAsmStr80);      {Returned substitution      }
```

where TYPE D is

```
AsmStr80 = String[80];
```

or in C,

```
pascal void LookUp(Ptr         PC,
                   LookupRegs   BaseReg,
                   long         Opnd,
                   char         *S);
```

These values are explained here:

PC
PC means pointer to instruction extension word or A-trap word in the buffer pointed to by the Disassembler's `FirstByte` parameter.

BaseReg
`BaseReg` determines the meaning of the `Opnd` value and supplies the base register for the "∂(An)", "BD,An", and "±∂" cases.

`BaseReg` may contain any one of the following values:

■ **Table F-4** Base register values

_A0_	= 0 ==> A0
_A1_	= 1 ==> A1
_A2_	= 2 ==> A2
_A3_	= 3 ==> A3
_A4_	= 4 ==> A4
_A5_	= 5 ==> A5
_A6_	= 6 ==> A6
_A7_	= 7 ==> A7
_PC_	= 8 ==> PC-relative (special case)
ABS	= 9 ==> Abs addr (special case)
TRAP	= 10 ==> Trap word (special case)

For absolute addressing (modes 70 and 71), `BaseReg` contains _ABS_.
For A-traps, `BaseReg` would contain _TRAP_.

Opnd
The contents of this `LongInt` is determined by the `BaseReg` parameter just described.

For `BaseReg` = _TRAP_ (A-traps)
> `Opnd` is the entire trap word. The high-order 16 bits of `Opnd` are zero.

For `BaseReg` = _ABS_ (absolute effective address)
> `Opnd` contains the (extended) 32-bit address specified by the instruction's effective address. Such addresses are generally used to reference low-memory globals on a Macintosh.

For `BaseReg` = _PC_ (PC-relative effective address)
> `Opnd` contains the 32-bit address represented by "*±∂" adjusted by the Disassembler's `DstAdjust` parameter.

For `BaseReg` = _An_ (effective address with a base register)
> `Opnd` contains the (sign-extended) 32-bit (base) displacement from the instruction's effective address.

In the Macintosh environment, a `BaseReg` specifying A5 implies either global data references or Jump Table references. Positive `Opnd` values with an A5 `BaseReg` thus mean Jump Table references, while a negative offset would mean a global data reference. Base registers of A6 or A7 would usually mean local data.

S
S is a Pascal string returned from Lookup containing the effective address substitution string or a trap name for A-traps. S is set to null *prior* to calling Lookup. If it is still null on return, the string is not used. If not null, then for A-traps, the returned string is used as a opcode string. In all other cases the string is substituted as shown in the above table.

Depending on the application, you have three choices on how to use the Disassembler and an associated Lookup procedure:

1. You can call just the Disassembler and provide your own Lookup procedure. In that case, you must follow the calling conventions discussed above.

2. You can provide NIL for the LookupProc parameter, in which case, *no* Lookup proc will be called.

3. You can first call InitLookup (described later in this appendix, a procedure provided with this unit) and pass the address of this unit's standard Lookup procedure when Disassembler is called. In this case, all the control logic to determine the kind of substitution to be done is provided for you and all that you need to provide are the routines to look up any or all of the following:

    ■ PC-relative references

    ■ Jump table references

    ■ Absolute address references

    ■ Trap names

    ■ References with offsets from base registers

The InitLookup procedure

```
PROCEDURE InitLookup(PCRelProc: UNIV Ptr;JTOffProc: UNIV Ptr;
        TrapProc: UNIV Ptr; AbsAddrProc: UNIV Ptr;IdProc: UNIV Ptr);
```

This procedure prepares for use of this unit's Lookup procedure. When the Disassembler is called and the address of this unit's Lookup procedure is specified, then for PC-relative, jump table references, A-traps, absolute addresses, and offsets from a base register, the associated level 1 Pascal procedure specified here is called (if it is not NULL—all five addresses are preset to NULL). The calls assume the following declarations for these procedures (see "Lookup" later in this appendix for further details):

```
PROCEDURE PCRelProc(Address: UNIV LongInt;
                VAR S:    UNIV DisAsmStr80);

PROCEDURE JTOffProc(A5JTOffset: UNIV Integer;
                VAR S:        UNIV DisAsmStr80);

PROCEDURE TrapNameProc(TrapWord: UNIV Integer;
                   VAR S:     UNIV DisAsmStr80);

PROCEDURE AbsAddrProc(AbsAddr: UNIV LongInt;
                   VAR S:    UNIV DisAsmStr80);
```

```
PROCEDURE IdProc(BaseReg:  LookupRegs;
                 Offset:   UNIV LongInt;
                 VAR S:    UNIV DisAsmStr80);
```

or in C,

```
pascal void PCRelProc(long Address, char *S)
pascal void JTOffProc(short A5JTOffset, char *S)
pascal void TrapNameProc(unsigned short TrapWord, char *S)
pascal void AbsAddrProc(long AbsAddr, char *S)
pascal void IdProc(LookupRegs BaseReg, long Offset, char *S)
```

◆ *Note:* InitLookup contains initialized data that requires initializing at load time. This of concern only to users with assembly main programs.

The Lookup procedure

```
PROCEDURE  Lookup(PC: UNIV Ptr;BaseReg: LookupRegs;Opnd:
          UNIV LongInt;VAR S: DisAsmStr80);
```

This is a standard Lookup procedure available for calls to the Disassembler. If you use this procedure, then you must call InitLookup prior to any calls to the Disassembler. This procedure performs all the logic to determine the type of lookup. For PC-relative, jump table references, A-traps, absolute addresses, and offsets from a base register, the associated level 1 Pascal procedure specified in the InitLookup call (if not NULL) is called.

This scheme simplifies the Lookup mechanism by allowing you to focus on the problems related to the application.

The LookupTrapName procedure

```
PROCEDURE LookupTrapName(TrapWord: UNIV Integer;VAR S: UNIV
DisAsmStr80);
```

This procedure allows conversion of a trap instruction (in TrapWord) to its corresponding trap name (in S). It is provided primarily for use with the Disassembler and its address may be passed to InitLookup above for use by this unit's Lookup routine. Alternatively, there is nothing prohibiting you from using it directly for other purposes or by some other lookup procedure.

◆ *Note:* The tables in this procedure make the size of this procedure about 9500 bytes. The trap names are fully spelled out in upper and lower case. ·

The ModifyOperand procedure

```
PROCEDURE ModifyOperand(VAR Operand: UNIV DisAsmStr80);
```

The procedure scans an operand string, that is, the null-terminated Pascal string returned by the Disassembler (null *must* be present here), and modifies negative hex values to negated positive value. For example, $FFFF(A5) would be modified to –$0001(A5). The operand to be processed is passed as the function's parameter, which is then edited "in place" and returned to the caller.

This routine is essentially a pattern matcher and attempts to modify only 2-, 4-, and 8- digit hex strings in the operand that "might" be offsets from a base register. If the matching tests are passed, the same number of original digits are output (because that indicates a value's size: byte, word, or long).

For a hex string to be modified, the following tests must be passed:

■ There must have been exactly 2, 4, or 8 digits. Only hex strings $XX, $XXXX, and $XXXXXXXX are possible candidates because that is the only way the Disassembler generates offsets.

■ The hex string must be delimited by a left parenthesis character," (" or a comma, " , ". The left parenthesis character allows offsets for $xxxx(An, ...) and $xx(An, Xn) addressing modes. The comma allows for the MC68020 addressing forms.

■ The "$x..." must *not* be preceded by a plus-or-minus sign, " ± ". This eliminates the possibility of modifying the offset of a PC-relative addressing mode always generated in the form "*±$xxxx".

■ The "$x..." must *not* be preceded by a pound sign, " # ". This eliminates modifying immediate data.

■ Value must be negative. Negative values are the only values modified. A value $FFFF is modified to –$0001.

The validMacsBugSymbol function

```
FUNCTION   validMacsBugSymbol(symStart: UNIV Ptr;limit: UNIV Ptr;
           symbol:StringPtr): StringPtr; C;
```

Check that the bytes pointed to by symStart represent a valid MacsBug symbol. The symbol must be fully contained in the bytes starting at symStart, up to but not including the byte pointed to by the limit parameter.

If a valid symbol is *not* found, then NULL is returned as the function's result. However, if a valid symbol is found, it is copied to symbol (if it is not NULL) as a null-terminated Pascal string, and return a pointer to where we think the *following* module begins. In the "old style" cases (see the following table) this will always be 8 or 16 bytes after the input `symStart`. For new style Apple Pascal and C cases this will depend on the symbol length, existence of a pad byte, and size of the constant (literal) area. In all cases, trailing blanks are removed from the symbol.

A valid MacsBug symbol consists of the characters '_', '%', spaces, digits, and upper and lower case letters in a format determined by the first two bytes of the symbol as follows:

1st byte range	2nd byte range	Byte length	Comments
$20 –$7F	$20 – $7F	8	Old style MacsBug symbol format
$20 –$7F	$20 – $7F	8	Old style MacsBug symbol format
$A0 – $FF	$20 –$7F	8	Old style MacsBug symbol format
$20 –$7F	$80 – $FF	16	Old style MacApp symbol ab==>b.a
$A0 – $FF	$80 – $FF	16	Old style MacApp symbol ab==>b.a
$80	$01 – $FF	n	n = 2nd byte (Apple Compiler symbol)
$81 – $9F	$00 – $FF	m	m = 1st byte & $7F (Apple Compiler symbol)

The formats are determined by whether bit 7 is set in the first and second bytes. This bit will removed when it is found OR'ed into the first and/or second valid symbol characters.

The first two formats in the above table are the basic "old-style" (pre-existing) MacsBug formats. The first byte may or may not have bit 7 set if the second byte is a valid symbol character. The first byte (with bit 7 removed) and the next 7 bytes are assumed to comprise the symbol.

The second pair of formats are also old-style formats, used for MacApp symbols. Bit 7 set in the second character indicates these formats. The symbol is assumed to be 16 bytes with the second 8 bytes preceding the first 8 bytes in the generated symbol. For example, 12345678abcdefgh represents the symbol abcdefgh.12345678.

The last pair of formats are reserved by Apple and generated by the MPW Pascal and MPW C compilers. In these cases the value of the first byte is always between $80 and $9F, or with bit 7 removed, between $00 and $1F. For $00, the second byte is the length of the symbol with that many bytes following the second byte (thus a maximum length of 255). Values $01 to $1F represent the length itself. A pad byte may follow these variable length cases if the symbol does not end on a word boundary. Following the symbol and the possible pad byte is a word containing the size of the constants (literals) generated by the compiler.

◆ *Note:* If `symStart` actually does point to a valid MacsBug symbol, then you can use `showMacsBugSymbol` to convert the MacsBug symbol bytes to a string that could be used as a DC.B operand for disassembly purposes. This string explicitly shows the MacsBug symbol encodings.

The endOfModule function

```
FUNCTION endOfModule(address: UNIV Ptr;limit: UNIV Ptr;symbol:
          StringPtr; VAR nextModule: UNIV Ptr): StringPtr; C;
```

This function checks to see if the specified memory address contains a `RTS, JMP (A0)` or `RTD #n` instruction immediately followed by a valid MacsBug symbol. These sequences are the only ones that can determine an end of module when MacsBug symbols are present. During the check, the instruction and its following MacsBug symbol must be fully contained in the bytes starting at the specified address parameter, up to, but not including, the byte pointed to by the limit parameter.

If the end of module is *not* found, then NULL is returned as the function's result. However, if a end of module is found, the MacsBug symbol is returned in symbol (if it is not NULL) as a null-terminated Pascal string (with trailing blanks removed), and the function returns the pointer to the start of the MacsBug symbol (that is, address+2 for `RTS` or `JMP (A0)` and address+4 for `RTD #n`). This address may then be used as an input parameter to `showMacsBugSymbol` to convert the MacsBug symbol to a Disassembler operand string.

Also returned in `nextModule` is where the *following* module is expected to begin. In the old-style cases (see `validMacsBugSymbol`) this will always be 8 or 16 bytes after the input address. For the new style, the Apple Pascal and C cases, this will depend on the symbol length, existence of a pad byte, and size of the constant (literal) area. See `validMacsBugSymbol` for a description of valid MacsBug symbol formats.

The showMacsBugSymbol function

```
FUNCTION  showMacsBugSymbol(symStart: UNIV Ptr;limit: UNIV Ptr;operand:
          StringPtr; VAR bytesUsed: INTEGER): StringPtr; C;
```

This function formats a MacsBug symbol as a operand of a DC.B directive. The first one or two bytes of the symbol are generated as $80+'c' if their high bits are set. All other characters are shown as characters in a string constant. The pad byte, if present, is also shown as $00.

This routine is called to check that the bytes pointed to by `symStart` represent a valid MacsBug symbol. The symbol must be fully contained in the bytes starting at `symStart`, up to but not including the byte pointed to by the limit parameter.

When called, showMacsBugSymbol assumes that symStart is pointing at a valid MacsBug symbol as validated by the validMacsBugSymbol or endOfModule routine. As with validMacsBugSymbol, the symbol must be fully contained in the bytes starting at symStart up to, but not including, the byte pointed to by the end parameter.

The string is returned in the 'operand' parameter as a null-terminated Pascal string. The function also returns a pointer to this string as its return value (NULL is returned only if the byte pointed to by the limit parameter is reached prior to processing the entire symbol— which should not happen if properly validated). The number of bytes used for the symbol is returned in bytesUsed. Due to the way MacsBug symbols are encoded, bytesUsed may not necessarily be the same as the length of the operand string.

A valid MacsBug symbol consists of the characters '_', '%', spaces, digits, and upper/lower case letters in a format determined by the first two bytes of the symbol as described in the validMacsBugSymbol routine.

Appendix G The Graf3D Library

GRAF3D IS A SET OF QUICKDRAW CALLS USED TO PRODUCE THREE-DIMENSIONAL GRAPHICS by providing a fixed-point interface to QuickDraw's integer coordinates. This appendix describes these routines and their use for both MPW Pascal and MPW C. ∎

Contents

Overview

The Graf3D routines provide several important features:

- A camera's-eye view. This allows you to set the point of view from which the observer sees the object independently from the coordinates of the object itself. The camera is set up with the ViewPort, LookAt, and ViewAngle procedures. You can set the focal length of the camera as if you had a choice of telephoto, wide-angle, or normal lenses.

- Three-dimensional clipping to a true pyramid. The apex of the pyramid is at the point of the camera eye, and the base of the pyramid is equivalent to the viewport. When you use the Clip3D function, only objects in front of the camera eye and within the pyramid are displayed on the screen.

- Two-dimensional point and line capability using Fixed type coordinates. Graf3D provides commands corresponding to the QuickDraw commands but using Fixed type coordinates instead of integers. With Fixed type coordinates you have a larger dynamic range for graphics calculations; with integer coordinates you get faster drawing time.

- Two-dimensional or three-dimensional rotation. You can rotate an object along any or all axes simultaneously, by using the Pitch, Yaw, and Roll procedures.

- Translation and scaling of objects in one or more axes simultaneously. *Translation* means movement anywhere in three-dimensional space. *Scaling* means shrinking or expanding.

How to use Graf3D

This section describes the language-specific preparations you need to make to use Graf3D with your MPW Assembler, MPW Pascal, or MPW C programs.

How to use Graf3D—MPW Assembler

To use Graf3D with MPW Assembler, do the following:

■ Include the file Graf3DEqu.a in your source text.

■ Link your assembly with the file {Libraries}Interface.o.

■ Set values in the Graf3D data structures and call the Graf3D routines from your program, using the equates in Graf3DEqu.a.

Throughout the rest of this appendix, Graf3D is described solely in MPW Pascal and MPW C notation. The Graf3d routines are implemented in MPW Pascal; Assembly-language programmers should call these routines by using Pascal calling conventions. For information on how to convert this notation into assembly-language calling conventions for stack-based routines, see the chapter "Using Assembly Language" in *Inside Macintosh*.

How to use Graf3D—MPW Pascal

To use Graf3D in MPW Pascal, do the following:

■ Include the declaration USES Graf3D in your source text.

■ Link your assembly program or object file with the file Interface.o.

■ Set values in the Graf3D data structures, and call the Graf3D routines from your program, following the information given in the section "Graf3D Data Types" that follows in this appendix.

How to use Graf3D—MPW C

To use Graf3D in MPW C, do the following:

1. Include these statements in your source text:
   ```
   #include <Types.h>
   #include <QuickDraw.h>
   #include <Graf3D.h>
   ```

2. Link your object file with the file {Libraries} Interface.o.

3. Set values in the Graf3D data structures and call the Graf3D routines from your program, following the information given in the section "Graf3D Data Types" that follows in this appendix.

Graf3D data types

Graf3D declares and uses these data types:

- `Fixed`
- `Point3D`
- `Point2D`
- `XfMatrix`
- `Port3DPtr`

The type `Fixed` is discussed in *Inside Macintosh*, Volume 1. The other types are discussed in this section. Examples of the calls are supplied in MPW Pascal and MPW C.

Point3D

`Point3D` contains three fixed-point number coordinates: x, y, and z. Graf3D uses x, y, and z for fixed-point number coordinates to distinguish between the h and v integer screen coordinates used by QuickDraw.

MPW Pascal

```
TYPE Point3D = RECORD
               x: Fixed;
               y: Fixed;
               z: Fixed
             END;
```

MPW C

```
typedef struct Point3D {
      Fixed x, y, z;
} Point3D;
```

Point2D

`Point2D` is just like a `Point3D` but contains only x- and y- coordinates.

MPW Pascal

```
TYPE Point2D = RECORD
               x: Fixed;
               y: Fixed
             END;
```

MPW C

```
typedef struct Point2D {
      Fixed x, y;
} Point2D;
```

XfMatrix

The `XfMatrix` is a 4x4 matrix of `Fixed` values used to hold a transformation equation.
Each transforming routine alters this matrix so that it contains the concatenated effects
of all transformations applied.

MPW Pascal

```
XfMatrix = ARRAY[0..3, 0..3] OF Fixed;
```

MPW C

```
typedef Fixed XfMatrix[4][4];
```

Macintosh® Programmer's Workshop Development Environment

Version 3.2

This package contains

2	Manuals	*Macintosh Programmer's Workshop Development Environment, Volume 1*
		Macintosh Programmer's Workshop Development Environment, Volume 2
1	Set of release notes	*MPW® 3.2 Release Notes*
5	Disks	*MPW Development Environment Installation Disk*, Disk 1 of 5
		MPW Development Environment, Version 3.2, Disk 2 of 5
		MPW Development Environment, Version 3.2, Disk 3 of 5
		MPW Development Environment, Version 3.2, Disk 4 of 5
		MPW Development Environment, Version 3.2, Disk 5 of 5
1	1-inch binder	
1	2-inch binder	
20	Tab dividers	
1	*MPW Survey*	

If you have any questions, please call

1-800-282-2732   (U.S.)
1-408-562-3910   (International)
1-800-637-0029   (Canada)

Port3DPtr

The type `Port3DPtr` contains all the state variables needed to map fixed-point number coordinates into integer screen coordinates.

MPW Pascal

```
Port3DPtr = ^Port3D;
Port3D    = RECORD
                GrPort: GrafPtr;
                viewRect: Rect;
                xLeft, yTop, xRight, yBottom: Fixed;
                pen, penPrime, eye: Point3D;
                hSize, vSize: Fixed;
                hCenter, vCenter: Fixed;
                xCotan, yCotan: Fixed;
                ident:    boolean;
                xForm:    XfMatrix
            END;
```

MPW C

```
typedef struct Port3D {
        GrafPtr     grPort;
        Rect        viewRect;
        Fixed       xLeft, yTop, xRight, yBottom;
        Point3D     pen, penPrime, eye;
        Fixed       hSize, vSize;
        Fixed       hCenter, vCenter;
        Fixed       xCotan, yCotan;
        char        filler;
        char        ident;
        XfMatrix    xForm;
} Port3D, *Port3DPtr;
```

■ **Table G-1**   Port3DPtr variables

Name	Description
GPort	Pointer to the grafPort associated with this Port3D
viewRect	Viewing rectangle within the grafPort; the base of the viewing pyramid
xLeft, yTop, xRight, yBottom	World coordinates corresponding to the viewRect
pen	Three-dimensional pen location
penPrime	Pen location transformed by the xForm matrix
eye	Three-dimensional viewpoint location established by ViewAngle
hSize, vSize	Half-width and half-height of the viewRect in screen coordinates
hCenter, vCenter	Center of the viewRect in screen coordinates
xCotan, yCotan	Viewing cotangents set up by ViewAngle, used by Clip3D
Ident	Boolean that allows the transformation to be skipped when xForm is an identity matrix
xForm	4x4 matrix that holds the net result of all transformations

Graph3D procedures and functions

Graf3D provides the following procedures and functions to establish a graphics environment and create drawings within it:

■ The InitGraf procedure

■ The Open3DPort procedure

■ The SetPort3D procedure

■ The GetPort3D procedure

■ The Move procedures

■ The Line procedures

■ The Clip3D function

■ The SetPoint procedures

Each procedure and function is described in this section in both MPW Pascal and MPW C.

The InitGraf3D procedure

The `InitGraf3D` procedure starts up Graf3D and initializes its data structures. *GlobalPtr* is a pointer to heap space for Graf3D data. Call `InitGraf3D` once and only once at the beginning of your program. Pass it at the address of a `Port3DPtr` (using the @ operator) that you have declared and reserved for use by Graf3D.

MPW Pascal

```
PROCEDURE InitGraf3D(GlobalPtr: Ptr);
```

MPW C

```
pascal void InitGrf3D(port)
     Port3DPtr *port;
```

The `InitGraf3D` function initializes the `Port3D` variable. Call this routine before doing Graf3D operations. Allocate space for a variable of type `Port3DPtr` (whose address is passed as a parameter to this function).

The Open3DPort procedure

The `Open3DPort` procedure initializes all the fields of a `Port3D` to their defaults, and makes that `Port3D` the current one. `Gport` is set to the currently open `grafPort`. These are the default values:

```
thePort3D := port;
port^.GPort := thePort;
ViewPort(thePort^.portRect);
WITH thePort^.portRect DO LookAt(left, top, right, bottom);
ViewAngle(0);
Identity;
MoveTo3D(0, 0, 0);
```

MPW Pascal

```
PROCEDURE Open3DPort(port: Port3DPtr);
```

MPW C

```
pascal void Open3DPort(port)
     Port3DPtr port;
```

The SetPort3D procedure

The SetPort3D procedure makes *port* the current Port3D and calls SetPort for that Port3D's associated grafPort. SetPort3D allows an application to use more than one Port3D and switch between them.

MPW Pascal

```
PROCEDURE SetPort3D(port: Port3Dptr);
```

MPW C

```
pascal void SetPort3D(port)
     Port3DPtr    port;
```

The GetPort3D procedure

The GetPort3D procedure returns a pointer to the current Port3D. This procedure is useful when you are using more than one Port3D and want to save and restore the current one.

MPW Pascal

```
PROCEDURE GetPort3D(VAR port: Port3DPtr);
```

MPW C

```
pascal void GetPort3D(port)
     Port3D       *port;
```

The Move procedures

Graf3D provides four Move procedures that move the pen in two or three dimensions without drawing lines. The fixed-point number coordinates are transformed by the xForm matrix and projected onto flat screen coordinates; then Graf3D calls QuickDraw's MoveTo procedure with the result.

MPW Pascal

```
PROCEDURE MoveTo2D(x, y: Fixed);
PROCEDURE MoveTo3D(x, y, z: Fixed);
PROCEDURE Move2D(dx, dy: Fixed);
PROCEDURE Move3D(dx, dy, dz: Fixed);
```

MPW C

```
pascal void MoveTo2D(x, y)
       Fixed x, y;
pascal void MoveTo3D(x, y, z)
       Fixed x, y, z;
pascal void Move2D(x, y)
       Fixed x, y;
pascal void Move3D(x, y, z)
       Fixed x, y, z;
```

The Line procedures

Graf3D provides four Line procedures that draw two- and three-dimensional lines from the current pen location. The LineTo2D and LineTo3D procedures stay on the same z-plane. The fixed-point number coordinates are first transformed by the xForm matrix, then clipped to the viewing pyramid, then projected onto the flat screen coordinates and drawn by calling QuickDraw's LineTo procedure.

MPW Pascal

```
PROCEDURE LineTo2D(x, y: Fixed);
PROCEDURE LineTo3D(x, y, z: Fixed);
PROCEDURE Line2D(dx, dy: Fixed);
PROCEDURE Line3D(dx, dy, dz: Fixed);
```

MPW C

```
pascal void LineTo2D(x, y)
      Fixed x, y;
pascal void LineTo3D(x, y, z)
      Fixed x, y, z;
pascal void Line2D(x, y)
      Fixed x, y;
pascal void Line3D(x, y, z)
      Fixed x, y, z;
```

The Clip3D function

The clip3D function clips a three-dimensional line segment to the viewing pyramid and returns the clipped line projected onto screen coordinates. clip3D returns true (nonzero) if any part of the line is visible. If no part of the line is within the viewing pyramid, clip3D returns false (zero).

MPW Pascal

```
FUNCTION Clip3D(src1, src2: Point3D; VAR dst1, dst2: Point):
          boolean;
```

MPW C

```
pascal short Clip3D(src1, src2, dst1, dst2)
      Point3D    *src1, *src2;
      Point      *dst1, *dst2;
```

The Set Point procedures

Graf3D provides two Set Point procedures. The setPt3D procedure assigns three fixed-point numbers to a Point3D. The setPt2D procedure assigns two fixed-point numbers to a Point2D.

MPW Pascal

```
PROCEDURE SetPt3D(VAR pt3D: Point3D; x, y, z: Fixed);
PROCEDURE SetPt2D(VAR pt2D: Point2D; x, y: Fixed);
```

MPW C

```
pascal void SetPt3D(pt3D, x, y, z)
      Point3D       *pt3D;
      Fixed         x, y, z;
pascal void SetPt2D(pt2D, x, y)
      Point2D       *pt2D;
      Fixed         x, y;
```

Setting up the camera

Procedures ViewPort, LookAt, and ViewAngle position the image in the grafPort,
aim the camera, and choose the lens focal length in order to map three-dimensional
coordinates onto the flat screen space. These procedures may be called in any order.

The ViewPort procedure

The ViewPort procedure specifies where to put the image in the grafPort. The
ViewPort rectangle is in integer QuickDraw coordinates and tells where to map the
LookAt coordinates.

MPW Pascal

```
PROCEDURE ViewPort(r: Rect);
```

MPW C

```
pascal void ViewPort(r)
      Rect  *r;
```

The LookAt procedure

The `LookAt` procedure specifies the fixed-point number x- and y-coordinates corresponding to the `viewRect`.

MPW Pascal

```
PROCEDURE LookAt(left, top, right, bottom: Fixed);
```

MPW C

```
pascal void LookAt(left, top, right, bottom)
     Fixed left, top, right, bottom;
```

The ViewAngle procedure

The `ViewAngle` procedure controls the amount of perspective by specifying the horizontal angle (in degrees) subtended by the viewing pyramid. Typical viewing angles are 0° (no perspective), 10° (telephoto lens), 25° (normal perspective of the human eye), and 80° (wide-angle lens).

MPW Pascal

```
PROCEDURE ViewAngle(angle: Fixed);
```

MPW C

```
pascal void ViewAngle(angle)
     Fixed angle;
```

The transformation matrix

Use the transformation (xForm) matrix to impose a coordinate transformation between the coordinates you plot and the viewing coordinates. Each of the transformation procedures concatenates a cumulative transformation onto the xForm matrix. Subsequent lines drawn are first transformed by the xForm matrix, then projected onto the screen as specified by ViewPort, LookAt, and ViewAngle.

The Identity procedure

The Identity procedure resets the transformation matrix to an identity matrix.

MPW Pascal

```
PROCEDURE Identity;
```

MPW C

```
pascal void Identity();
```

The Scale procedure

The Scale procedure modifies the transformation matrix to shrink or expand by *xFactor*, *yFactor*, and *zFactor*. For example,

```
Scale(X2Fix(2.0), X2Fix(2.0), X2Fix(2.0))
```

doubles the size of whatever you draw.

MPW Pascal

```
PROCEDURE Scale (xFactor, yFactor, zFactor: Fixed);
```

MPW C

```
pascal void Scale(xFactor, yFactor, zFactor)
     Fixed xFactor, yFactor, zFactor;
```

The Translate procedure

The `Translate` procedure modifies the transformation matrix so as to displace by *dx*, *dy*, and *dz*.

MPW Pascal

```
PROCEDURE Translate(dx, dy, dz: Fixed);
```

MPW C

```
pascal void Translate(dx, dy, dz)
      Fixed dx, dy, dz;
```

The Pitch procedure

The `Pitch` procedure modifies the transformation matrix so as to rotate *xAngle* degrees around the *x*-axis. A positive angle rotates clockwise when looking at the origin from positive *x*.

MPW Pascal

```
PROCEDURE Pitch(xAngle: Fixed);
```

MPW C

```
pascal void Pitch(xAngle)
      Fixed xAngle;
```

The Yaw procedure

The `Yaw` procedure modifies the transformation matrix to rotate *yAngle* degrees around the *y*-axis. A positive angle rotates clockwise when looking at the origin from positive *y*.

MPW Pascal

```
PROCEDURE Yaw(yAngle: Fixed);
```

MPW C

```
pascal void Yaw(yAngle)
      Fixed yAngle;
```

The Roll procedure

The Roll procedure modifies the transformation matrix so as to rotate *zAngle* degrees around the z-axis. A positive angle rotates clockwise when looking at the origin from positive z.

MPW Pascal

```
PROCEDURE Roll(zAngle: Fixed);
```

MPW C

```
pascal void Roll(zAngle)
      Fixed zAngle;
```

The Skew procedure

The Skew procedure modifies the transformation matrix so as to skew *zAngle* degrees around the z-axis. Skew changes only the x-coordinate; the result is much like the slant that QuickDraw gives to italic characters. (Skew(15.0) makes a reasonable italic.) A positive angle rotates clockwise when looking at the origin from positive z.

MPW Pascal

```
PROCEDURE Skew(zAngle: Fixed);
```

MPW C

```
pascal void Skew(zAngle)
      Fixed zAngle;
```

The Transform procedure

The `Transform` procedure applies the `xForm` matrix to *src* and returns the result as *dst*. If the transformation matrix is `Identity`, *dst* will be the same as *src*.

MPW Pascal

```
PROCEDURE Transform (src: Point3D; VAR dst: Point3D);
```

MPW C

```
pascal void Transform(src, dst)
     Point3D     *src, *dst;
```

Appendix H **Object File Format**

THIS APPENDIX IS ADDRESSED TO PROGRAMMERS who are writing compilers or assemblers to run under MPW 3.0. ■

Contents

About object file records

Object file format describes the structure of MPW object files. These files are created by various language processors (such as MPW Assembler, MPW Pascal, and MPW C) and the MPW librarian (Lib).

An object file consists of a sequence of object file records. The records described in the remainder of this appendix are located in the data fork of the object file. MPW object files have file type 'OBJ ' and creator 'MPS '.

◆ *Note:* The MPW linker validates only the file type, since other applications than MPW may create MPW-compatible object files.

At the present time the linker tools do not use the resource fork. Although the current versions of Link, Lib, and DumpObj will ignore resource fork information in object files, Apple may specify resources for object files at some time in the future.

There are currently 20 types of object file records, numbered consecutively from 0 to 19. (Future record types are not guaranteed to have consecutive record numbers).

Code records:

■ The first record in the file must be a **First** record.

■ The last record in the file must be a **Last** record.

■ One-byte **Pad** records are used to maintain word alignment.

■ **Comment** records allow comments to be included in the file.

■ **Dictionary** records associate names with unique IDs.

■ **Module** records define code and data modules.

■ **EntryPoint** records define entry points in code and data modules.

■ **Size** records specify the module sizes.

■ **Contents** records specify the contents of modules.

■ **Reference** and **ComputedReference** records specify locations in modules that contain references to other modules or entry points.

Symbolic records:

- **Filename** records specify source filenames and modification dates.

- **SourceStatement** records specify correspondence between module offsets and source statements.

- **ModuleBegin** and **ModuleEnd** records declare named scopes (typically associated with units, files, or functions).

- **BlockBegin,** and **BlockEnd** records declare unnamed or "block" scopes contained within modules.

- **LocalID** records declare identifiers scoped within modules or blocks.

- **LocalLabel** records specify correspondence between generated code, source statements, and label identifiers.

- **LocalType** records define type information for local identifiers and functions.

A **module** is a contiguous region of memory that contains code or static data. A module is the smallest unit of memory that is included or removed by the linker. An **entry point** is a location (offset) within a module. (The module itself is treated as an entry point with offset zero.) A **segment** is a named collection of modules.

◆ *Note:* The jump table (described in I*nside Macintosh, Volume II*) is considered to be code.

All modules, entries, segments, and symbolic records have a unique 16-bit ID that is assigned by the compiler, assembler, or librarian. An **ID** is an object file-related number that identifies a module, entry point, segment, or other entity within a single object file.

◆ *Note:* Older versions of the MPW linker support only ID values in the range 1...16,383. The current version accepts ID values in the range 1...65,534 (values 0 and 65,535 are reserved). If your compiler needs to be compatible with MPW 2.0, then you should restrict the range of IDs accordingly.

◆ *Note:* The linker is faster and more efficient if the compiler allocates IDs sequentially from 1.

Modules and entry points may be local or external.

■ A **local** module, entry point, or segment can be referenced only from within the file where it is defined

■ An **external** module, entry point, or segment can be referenced from different files. In addition to an ID, each external module or entry point defined or referenced in an object file must also have a unique name (a string identifier) that identifies it across files.

If an ID has a name, that name is specified in a **dictionary record.**

If no dictionary entry exists for it, an ID is considered **anonymous.**

Local modules and entries need not have unique names, and an external segment may have the same name as an external module or entry point.

◆ *Note:* Although the names need not be unique, the IDs of these different objects must be unique. There must be multiple dictionary entries even though the names are the same. If symbolic debugging records are generated, then **ModuleBegin** records that correspond to **Module** records must also share the same **ID.**

A segment is declared implicitly by specifying a segment ID in a code-type Module record.

At any given point in an object file there can be one current code module and one current data module. The beginning of a new code or data module is indicated by a Module record. The current code and data modules are further defined by Entry Point, Size, Contents, Reference, and Computed-Reference records—these records can occur in any order after the Module record.

In each of these **intra-module** records, a flag bit indicates whether the record refers to the code or the data stream, permitting the interleaving of code and data records by compilers. Code and data may be arbitrarily interleaved. For instance, the record sequence:

```
Module(Code, ID=1)
```
Contents, Size, EntryPoint, and reference records for module 1

```
Module(Data, ID=2)
```
Contents, Size, EntryPoint, and reference records for modules 1 and 2

```
Module(Code, ID=3)
```
Contents, Size, EntryPoint, and reference records for modules 2 and 3

```
Module(Code, ID=4)
```
Contents, Size, EntryPoint, and reference records for modules 2 and 4

declares three code modules and one data module; data module 2's scope extends until the next data module, across an arbitrary number of code modules.

Scoping of symbolic information

All symbolic records contain a parent ID field that specifies the scope to which the record applies. These records may be emitted in any order, in any part of the object file. Whether the code-generating and the symbolic records are interleaved, nested, or completely separated is left to the discretion of the language implementor. (However, source statement records for a particular module must be written in order by increasing source offset.)

Executable objects (functions and procedures) are scoped lexically, while variables are scoped according to their visibility. To see why this is so, consider the following example in C (Pascal UNITs are similar):

```
/* example.c */
static int static_var;
int public_var;
static local_func() { ... }
public_func() { ... }
```

If 'public_func' were contained by the root scope, then it would be impossible to access any file-level static variables (such as 'static_var') from within a breakpoint in the function. Even though the Module record for 'public_func' has its external bit set, the ModuleBegin for the function must specify the source file as the parent scope.

On the other hand, the variable 'public_var' must be visible to procedures outside the file scope, so it is necessary to specify the root as its parent.

The records generated for the above example might be:

```
First(version=3)
Dictionary(1, "example.c")
Filename(1, modificationDate)

Dictionary(2, "example.c")
ModuleBegin(moduleID=2, parentID=0, fileID=1, kind=Unit)
 Dictionary(3, "static_var")
 LocalID(parentID=2, fileID=1, ID=3, type, etc.)

 Dictionary(4, "public_var")
 LocalID(parentID=0, fileID=1, ID=4, type, etc.)

 Dictionary(5, "local_func")
 ModuleBegin(moduleID=5, parentID=2, fileID=1, kind=Function)
 ModuleEnd(moduleID=5)

 Dictionary(6, "public_func")
 ModuleBegin(moduleID=6, parentID=2, fileID=1, kind=Function)
 ModuleEnd(moduleID=6)
ModuleEnd(moduleID=2, fileOffset)

Module(ID=5)
Contents, references, and entry points for module 5
Likewise, source statement information

Module(ID=6, flags=external)
Contents, references, and entry points for module 6
Likewise, source statement information

Last
```

The symbolic records may appear in any order (ModuleEnds preceding ModuleBegins, if necessary), interspersed with the nonsymbolic records (which *do* have order dependencies). There are a few items of interest in this example:

- A parent ID of zero indicates the root.

- The ID referred to by a Filename record cannot also be referred to by a ModuleBegin record. Languages (such as C) which have file scoping will need to produce two Dictionary records that have the same name but different IDs: one for the filename, the other for the file-level scope.

- A point of fine style: The linker is more efficient when multiple entries are emitted in the same Dictionary record. Many of the symbolic records also allow multiple definitions in the same record, and compiler writers should make use of this facility.

- Even though it may be necessary to emit data Module records for variables, it is incorrect to emit ModuleBegin records for the data modules; use LocalID records instead.

ModuleBegin implementation/declaration semantics

The terms *implementation* and *declaration* refer to ModuleBegin records with the **isDeclaration** bit set at, respectively, zero or one.

If a module has only an implementation, the linker assumes the declaration and implementation source locations are the same. The first declaration encountered is used, except that a declaration in the same object file as the first implementation will override any previous declaration. Any declarations or implementations following the first implementation are ignored (anything contained in such an ignored scope is also ignored). It is legal to have a declaration without an implementation or a Module.

Module scope information is supplied solely by implementation records; declaration records may be nested, but the nesting is ignored since the declaration information does not affect mapping between the source and the executable code.

Local variables (such as parameter variables) and types should be attached to either the implementation or declaration, but not to both.

Record type notation

This section contains important information about the documentation conventions used in the record descriptions that follow. Each record type is represented by a diagram such as the following:

The first box illustrates the record. Each square block represents a byte. The first byte indicates the record type, in this case, 10. The Flags byte is expanded in the second box. The Record Size is a signed, 16-bit integer that indicates the total length of the record (including the record type byte, flags byte, and record size field). Hence, any one object file record is limited to 32767 bytes. (This is not a limit on the size of the module, because partial contents can be placed in several records.)

The second box represents the flag bits. In this example, they are interpreted as follows:

Bit	Meaning
0	0 indicates code, and 1 indicates data
1, 2	Must be 0
3	0 indicates short, and 1 indicates long
4-5	0 indicates 32 bits, 1 indicates 16 bits, and 2 indicates 8 bits
6	Must be 0
7	1 indicates a difference computation

◆ *Note:* All unspecified bits must be zero.

In the remainder of this document, names in bit-fields will be specified in numeric order. For instance, in the case of the text `'local/extern'`, the tags `local` and `extern` are understood to be respectively zero and one.

The records have been defined so that:

- All 16-bit and 32-bit fields are word-aligned in the file.

- Fixed-size records do not have a Record Size field.

- All variable-length records have a Record Size field.

Object file records

This section describes and diagrams each object file record.

Pad record

A pad record is a single byte that is always zero.

```
┌───────┐
│       │
│   0   │
│       │
└───────┘
```
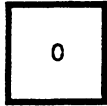
In order to maintain word alignment, a Pad record follows any record whose length is an odd number of bytes. (Other than pad records, all records are word-aligned.)

First record

The first record in an object file must be a First record.
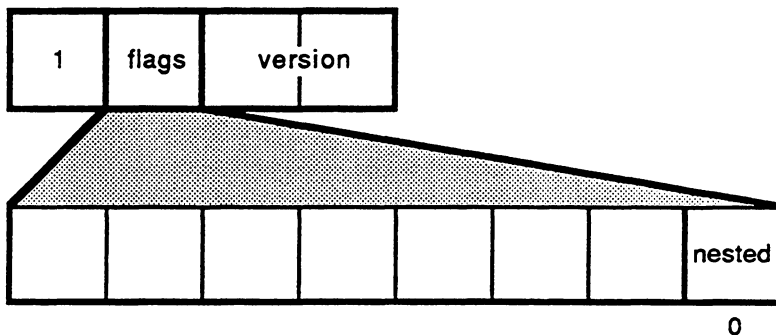
If the nested bit in the flags field is one, then the linker interprets all references to undefined ID-name pairs as external references. If the nested bit is zero, the linker will try to match the name of an undefined symbol with a local name before treating the undefined symbol as external.

The version field contains a version number:

Version number	Meaning
1	Nonsymbolic MPW 2.0 OMF file
2	Nonsymbolic MPW 3.0 OMF file
3	OMF file containing symbolic information

Last record

The last record in an object file must be a Last record.

```
┌───────┬───────┐
│   2   │   0   │
└───────┴───────┘
```
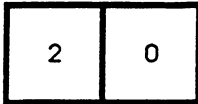
Comment record

A comment record allows comments to be included in an object file.

```
┌─────┬─────┬─────────────┬──────────────┐
│  3  │  0  │ record size │   >> comments >> │
└─────┴─────┴─────────────┴──────────────┘
```

The record size field specifies the total number of bytes in the record.

Dictionary record

A dictionary record associates a name with an ID (or several names with several IDs).
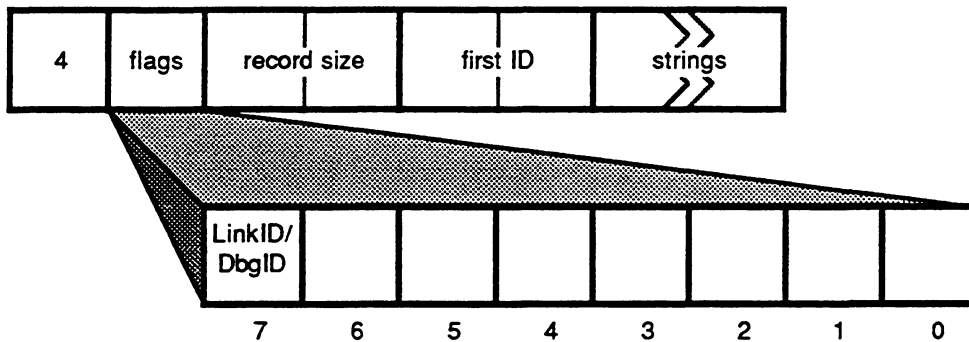
At most one dictionary record may appear for a given ID in a single object file.

The record size field specifies the total number of bytes in the record.

The strings field contains one or more names, each of which is preceded by an unsigned length byte.

The first name in the strings field is associated with the ID given in the FirstID field. The second name is associated with FirstID + 1, and so on.

The dictionary record for an ID must appear before the module or entry-point record that defines the ID, but need not appear before reference or computed-reference records that refer to the ID. If an ID has no dictionary record or has a name with a length of zero, it's considered anonymous.

The LinkID/DbgID flag bit is used to differentiate symbolic debugging identifiers from code-generating identifiers. When the linker is not linking symbolically it ignores dictionary records with this bit set, reducing the Link's memory requirements.

Module record

A Module record associates an ID with a module, and establishes that module as the "current" code or data module. All Entry-Point, Size, Contents, Reference, and Computed-Reference records combine to define a code or data module.
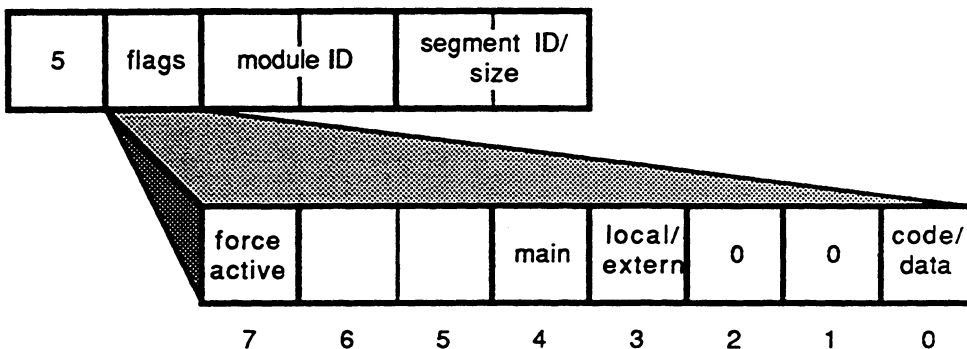
Modules may contain either code or data:

- For code modules, the segment ID field specifies the segment in which the code is placed. Segments may be named or anonymous. Named segments are treated as external; anonymous segments are local. (If the segment is named, the dictionary record specifying the name must appear before the segment ID can be used in a Module record.)

- For data modules, a nonzero size field specifies the size of the module. In this case Size or Contents records are unnecessary. (The size of a module can also be specified by a Size record, or implicitly specified by the offset of the last byte in a Contents record.)

Modules may be either local or external. (Local modules may be anonymous.)

A code module flagged as main becomes the execution starting point of the program. A data module flagged as main becomes the main program data area, just below the location pointed to by A5. At most one main code module or entry point and one main data module may appear in an object file.

If a code or data module has the force-active bit set, then the linker will not strip that module even though it is not referenced by any other module and is not the main module.

References to a module are considered to be references to the first byte of the module.

◆ *Note:* The linker ensures that modules are aligned on word or longword boundaries.

Entry-Point record

An Entry-Point record declares an entry-point ID. The entry point is in the current code or data module, as indicated by bit 0 of the flags field.
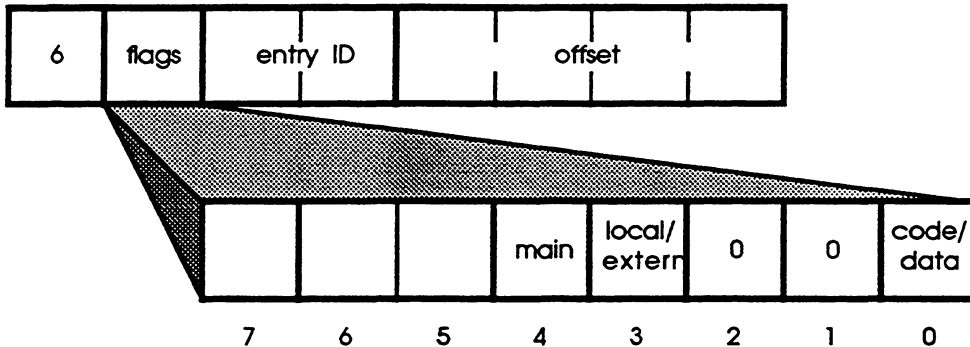
The offset field gives the byte offset of the entry point relative to the beginning of the module. The offset of an entry point may be outside the module (for example, a virtual base for an array).

An entry point may be defined for either a code or a data module. Entry points may be either local or external. (Local entry points may be anonymous.) A code entry point flagged as main becomes the execution starting point of the program. At most one main code module or entry point may appear in an object file.

Size record

A Size record specifies the size of the current code or data module in bytes.

7	flags	module size			

							code/ data
7	6	5	4	3	2	1	0
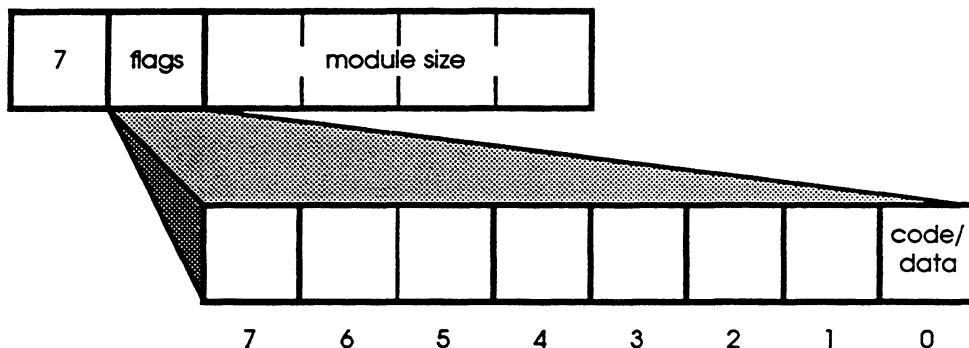
The size of a module may also be specified in a Contents record, or (for data modules) in the Module record. If more than one size is specified, the largest size given is taken as the size of the module.

Contents record

Contents records specify the contents of the current code or data module.

8	flags	record size	[offset]	[repeat]	contents

			repeat	compl/ partial	0	0	code/ data
7	6	5	4	3	2	1	0
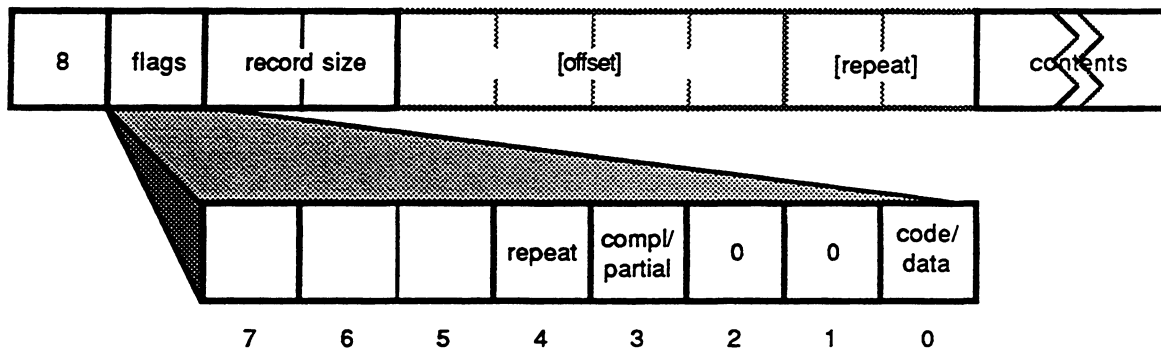
The record size field specifies the total number of bytes in the record.

Either complete or partial contents may be specified. If partial contents are specified, the first four bytes of the contents field specify the offset of the contents from the beginning of the module.

The contents may be either the bytes to be placed in the module, or a 2-byte repeat count followed by the bytes to be repeated. (If both an offset and a repeat count are specified, the offset comes first.)

Multiple Contents records per module are permitted, in any order. The offset of the last byte for which contents are specified determines the module's total size. (Size specifications may also appear in the Module record, and in Size records—if more than one size is specified, the largest size given is taken as the size of the module.)

Reference record

A Reference record specifies a list of references to an ID. The references are from the current code or data module, and may be to either code or data.

9	flags	record size	ID	offsets

	A5-relative	0	0	32/16	short/long	0	0	code/data
	7	6	5	4	3	2	1	0
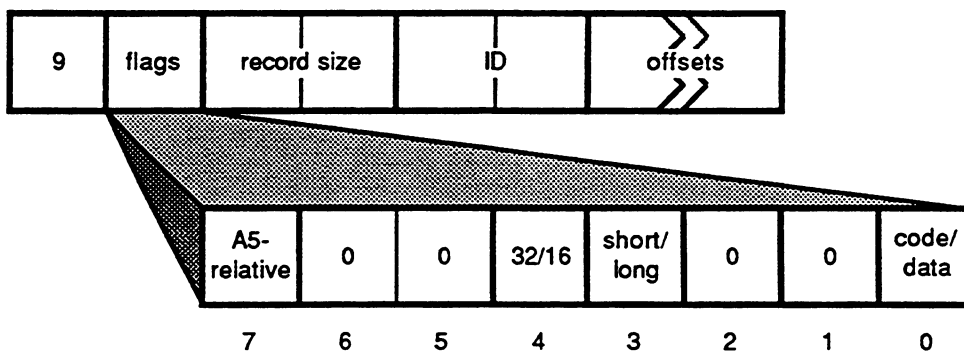
The record size field specifies the total number of bytes in the record.

The ID field specifies the module or entry point being referenced.

The offsets field specifies a list of byte offsets from the beginning of the current code or data module. These offsets may be either short (16 bits) or long (32 bits). The location modified may be either 32 or 16 bits. Multiple references to the same or overlapping locations are permitted. References from code may indicate instruction editing (that is, whether an offset is A5- or PC-relative).

References fall into the four categories described here:

- **Code-to-code references:** If the A5-relative flag is 1, the A5-relative offset of a jump-table entry associated with the specified module or entry is added to the specified location. No instruction editing is performed.

If the **A5-relative** flag is 0, the linker selects either PC-relative or A5-relative addressing. The immediately preceding 16-bit word must contain a JSR, JMP, LEA, or PEA instruction, and is modified to indicate either PC-relative or A5-relative addressing. If the referenced module or entry point and the current code module are in the same segment, the PC-relative offset of the module or entry point is added to the contents of the specified location. If they are in different segments, the A5-relative offset of a jump-table entry associated with the specified module or entry is added to the specified location.

■ **Code-to-data references:** The A5-relative flag must be 1 for code-to-data references. The A5-relative offset of the specified data module or entry is added to the contents of the specified location. No instruction editing is performed. The location may be either 16 or 32 bits. (32-bit A5-relative addressing is available for the MC68020, but not for the MC68000.)

■ **Data-to-code references:** If the A5-relative flag is 1, the A5-relative offset of a jump-table entry is added to the specified location, which may be either 16 or 32 bits.

If the A5-relative flag is 0, the memory address of a jump-table entry associated with the specified module or entry is added to the contents of the specified location, which must be 32 bits. (Note that this requires a run-time operation that adds the actual value of A5 to the A5-relative offset.)

■ **Data-to-data references:** If the A5-relative flag is 1, the A5-relative offset of the module or entry is added to the specified location, which may be either 32 or 16 bits.

If the A5-relative flag is 0, the memory address of the specified module or entry is added to the contents of the specified location, which must be 32 bits. (Note that this requires a run-time operation that adds the actual value of A5 to the A5-relative offset.)

	A5 = 0	A5 = 1
Code-to-Code	Edit instruction† Force PC-rel if in same segment, otherwise add JT-offset. **16**	Add JT-offset. Force PC-relative for non-'CODE' links†. **32/16**
Code-to-Data	Not allowed	Add A5-offset of data. **32/16**
Data-to-Code	Add JT offset of code, requires load-time addition of A5. **32**	Add JT-offset of code. **32/16**
Data-to-Data	Add A5-offset of data, requires load-time addition of A5. **32**	Add A5-offset of data. **32/16**

† Edited or forced instructions must be JMP, JSR, LEA or PEA.

32-bit code-to-code A5=1 references are possible in applications but not non-'CODE' links, because the instruction has to be forced PC-relative for a nonapplication link and 32-bit references cannot be edited.

This MPW Assembly language example exercises all possible modes of fixups except 32-bit code-to-code A5=6 which cannot easily be shown in MPW Assembler. Note that further instruction editing is done by the linker (for instance, the PC-relative JSR to PROC2 below will be forced A5-relative when the linker realizes that PROC2 is in a different segment).

```
       SEG    'SEG1'
PROC   MAIN
       IMPORT       PROC1:CODE, PROC2:CODE, DATA1:DATA
       IMPORT _DATAINIT  ;
       JSR    _DATAINIT   ; DO DATA INITIALIZATION

       CODEREFS FORCEJT   ; FORCE A5-RELATIVE:
       JSR    PROC1       ; CODE-TO-CODE, A5=1, SAME SEGMENT
       JSR    PROC2       ; CODE-TO-CODE, A5=1, DIFFERENT SEGMENT
       LEA    DATA1,A0    ; CODE-TO-DATA, A5=1

       CODEREFS NOFORCEJT; FORCE PC-RELATIVE:
       JSR    PROC2       ; CODE-TO-CODE, A5=0, DIFFERENT SEGMENT
                         ;   (FORCED A5-RELATIVE BY LINKER)
       JSR    PROC1       ; CODE-TO-CODE, A5=0, SAME SEGMENT
       ENDMAIN


DATA1  RECORD
       IMPORT PROC1:CODE, DATA2:DATA

       DATAREFS RELATIVE ; FORCE A5-RELATIVE
       DC.L   PROC1       ; DATA-TO-CODE, A5=1, 32-BIT THROUGH A5
       DC.W   PROC1       ; DATA-TO-CODE, A5=1, 16-BIT THROUGH A5
       DC.L   DATA1       ; DATA-TO-DATA, A5=1, 32-BIT THROUGH A5
       DC.W   DATA1       ; DATA-TO-DATA, A5=1, 16-BIT THROUGH A5

       DATAREFS ABSOLUTE ; FORCE ABSOLUTE
       DC.L   PROC1       ; DATA-TO-CODE, A5=0, 32-BIT
       DC.L   DATA1       ; DATA-TO-DATA, A5=0, 32-BIT
       ENDR               ;

PROC1  PROC   EXPORT
       ENDPROC

       SEG    'SEG2'
PROC2  PROC   EXPORT
       ENDPROC
       END
```

The "load-time" addition of A5 is performed by the procedure '_DATAINIT', which appears in the library '{Libraries}Runtime.o'.

The Code-to-Code and Code-to-Data reference modes have obvious utility. The reason that non-A5-relative Code-to-Data references are disallowed is that there is no mechanism for fixing up code on the Macintosh, which would otherwise have to be done every time a segment containing such a reference were loaded.

C's pointer initialization makes use of the 32-bit Data-to-Code and Data-to-Data reference modes, as in the following examples:

```
extern int func();
int (*fp)() = func;     /* Data-to-Code, 32-bit, load-time addition of A5 */
int (**pfp) = &fp;      /* Data-to-Data, 32-bit, load-time addition of A5 */
```

The A5-relative Data-to-Code and Data-to-Data reference modes can be used for saving space if the application has a large number of pointers to data or code (a dispatch table, for instance).

Computed-Reference record

A Computed-Reference record specifies a list of computed references based on two specified IDs.
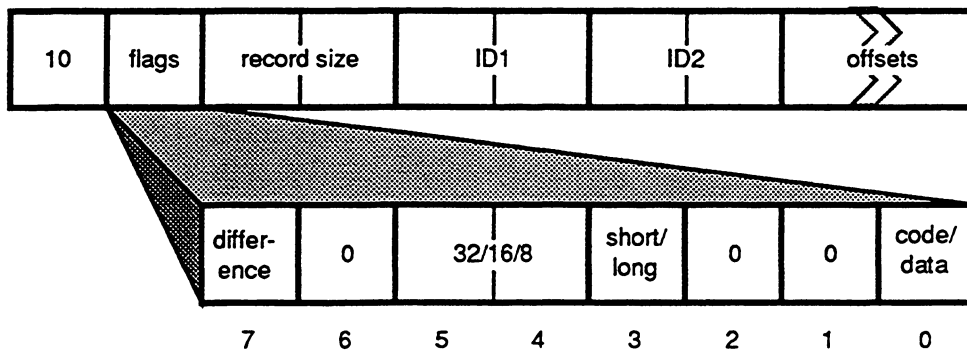
The record size field specifies the total number of bytes in the record. The references are from the current code or data module, and may be to either code or data.

The ID1 and ID2 fields specify the modules or entry points being referenced. If ID1 specifies a code reference, ID2 must also be a code reference in the same segment—if ID1 is a data reference, ID2 must also be a data reference.

The only computation provided is difference (that is, bit 7 must be set).

The offsets field specifies a list of byte offsets from the beginning of the current code or data module. These offsets may be either short (16 bits) or long (32 bits). The location modified may be either 32, 16, or 8 bits (a 0 in bits 4 and 5 indicates 32, 1 indicates 16, and 2 indicates 8). No instruction editing is performed.

The value of the address of ID1 minus the address of ID2 is added to the contents of the specified location. Multiple references to the same or overlapping locations are permitted.

Filename record

The Filename record associates a file object with a modification date.

11	0	file ID	modification date

The **file ID** associates a dictionary ID with a filename.

The modification date is the file's modification date. This is used by the debugger to help verify that the source file being displayed corresponds to the object file.

Although the fileID may be used by other records (such as source statements) prior to the appearance of the filename record, a Filename record must exist in the file for every fileID encountered. In addition, a dictionary ID must exist for the fileID; that is, files cannot be anonymous.

Source Statement record

The Source Statement record specifies the correspondence between generated code and source statements. The debugger uses this information to display source as a function of code location. The meaning of "statement" is defined by the language and the compiler's author.
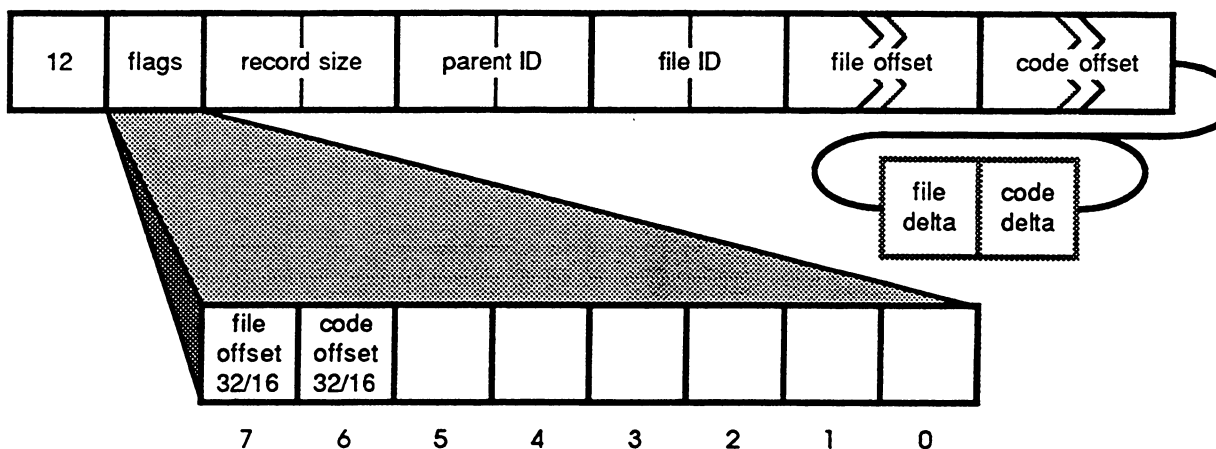
The record size field specifies the total number of bytes in the record.

The file ID associates a dictionary ID with a filename.

The parent ID specifies the scoping entity containing the statement.

The file offset and code offset specify the source file offset and code or data module offset for the first statement specified by this source statement record. These fields may be either 16 or 32 signed values. The file offset is the 0-relative byte offset in the file specified by the file ID. The code offset is the byte offset from the beginning of the code or data module for the first byte of code or data corresponding to the statement whose offset is specified by the file offset.

Each additional statement following the one specified by the file and code offset fields is specified by a file delta and code delta field. The deltas represent the *difference* between adjacent file and code offsets starting with the one specified by the file and code fields. These deltas are in the range from 0 to 255.

If the subsequent statement cannot be expressed with these offsets, then a new Source Statement record should be emitted with new beginning offsets.

All of the Source Statement records for a module must be emitted in order of increasing source code offset.

ModuleBegin record

The ModuleBegin record supplies symbolic information for a module.
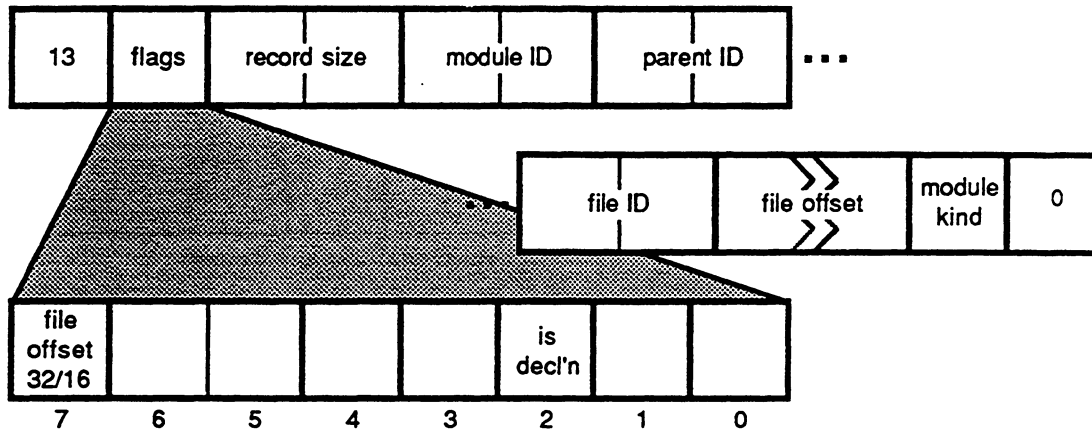
The isDeclaration bit in the flags field provides a way to specify source location information for a module's declaration, if the module's declaration and implementation are separated in the source code (such as a Pascal FORWARD or INTERFACE declaration).

The record size field specifies the total number of bytes in the record.

The module ID associates a dictionary ID with the record. This ID must be the same value used in the Module record. It is through this ID that the connection is made between the debugger object file stream for a module and the standard object module stream.

◆ *Note:* There doesn't have to be a Module record associated with the ModuleBegin. If there isn't a module with the same ID then the scope declared by the ModuleBegin is treated as a wrapper that doesn't have any code, but that can contain other symbolic objects. This is usually the case when the module-kind field is unit.

The parent ID specifies the scoping entity that contains the module. An ID of zero indicates global scope. This field is ignored if the isDeclaration bit is set (parentage information is not extracted from a declaration).

The file ID associates a dictionary ID with a filename.

The file offset specifies the source file offset for the module header. This field may be either 16 or 32 bits and is the 0-relative byte offset in the file specified by the file ID.

The module kind indicates the kind of the module as follows:

Kind	Description
0	none
1	*reserved*
2	unit
3	procedure
4	function
5	data module
6...15	*reserved*

A Pascal program's `program` level should be treated as a unit (module kind 2).

The byte following the module kind field is reserved.

ModuleEnd record

The ModuleEnd record is associated with a ModuleBegin record by the moduleID field.
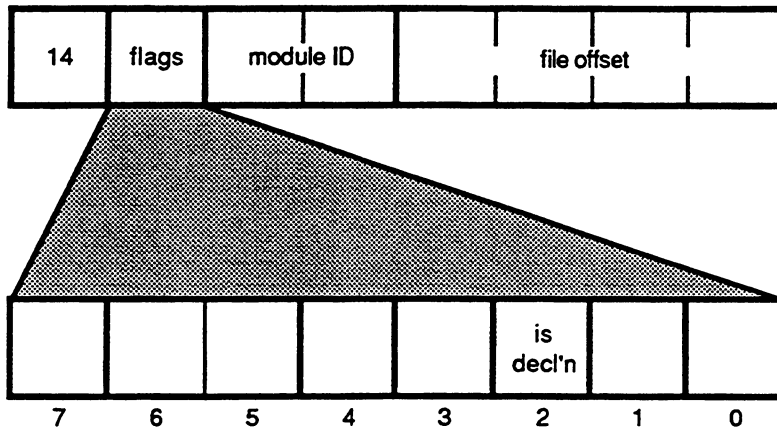
14	flags	module ID	file offset

					is decl'n		
7	6	5	4	3	2	1	0

The file offset specifies the last byte in the source file that is to be considered part of the module. The isDeclaration bit specifies whether the file offset reflects the end of the module's implementation or declaration.

BlockBegin record

A BlockBegin record declares a nested scope. Usually these scopes don't have names, but it's OK to associate a dictionary ID with the block scope *for symbolic naming purposes only;* the ID can't also be associated with another object (such as an entry point).
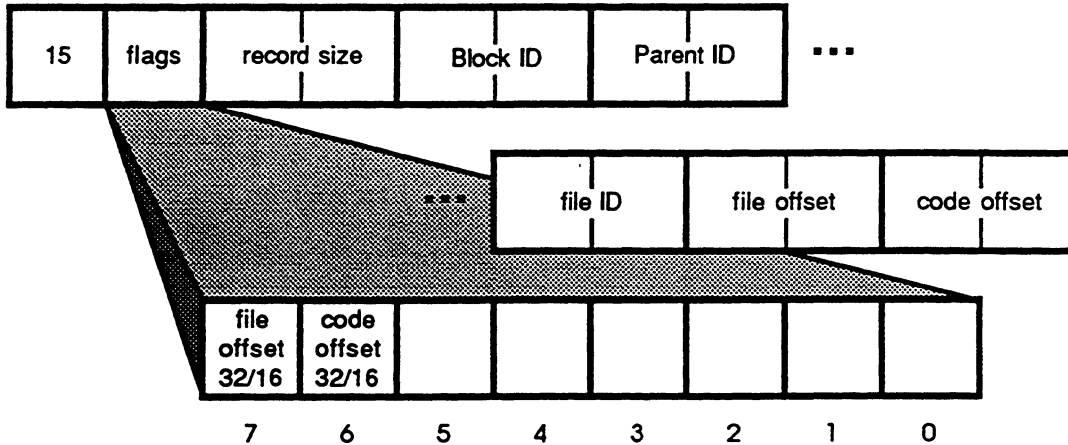
15	flags	record size	Block ID	Parent ID	•••

•••	file ID	file offset	code offset

file offset 32/16	code offset 32/16						
7	6	5	4	3	2	1	0

The record size field specifies the total number of bytes in the record.

The parent ID specifies the scoping entity (such as the block or module) containing the block. It's illegal to specify a parent of zero, or to specify a parentage chain that doesn't (eventually) include a {ModuleBegin, Module} pair; there must be a linkable object in the chain.

The file ID and file offset specify the source file and source file offset of the first statement within the block.

The code offset specifies the offset of the first instruction in the block. The offset is relative to the module that the block appears in, and does not depend on any intervening scopes between the BlockBegin and the module in which it appears.

A Block scope can't contain another ModuleBegin scope.

BlockEnd record

The BlockEnd record indicates the end of a block. There must be a block end record for each BlockBegin record.

16	0	BlockID	file offset	code offset

The file offset specifies the last byte in the source file that is to be considered part of the block.

The code offset specifies the offset within the containing module of the first instruction not to be treated as part of the block.

Local Identifier record

Local Identifier records specify formal parameters to procedures and functions as well as local identifiers (and their types) declared within modules or blocks.

The diagram fields (top record):

| 17 | flags | record size | parent ID | file ID | file offset >>> |

| file offset 32/16 | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Second record:

| ID | type | File Delta | kind | count | offset data >>> |

| offset size | refer-ence | value | storage class |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
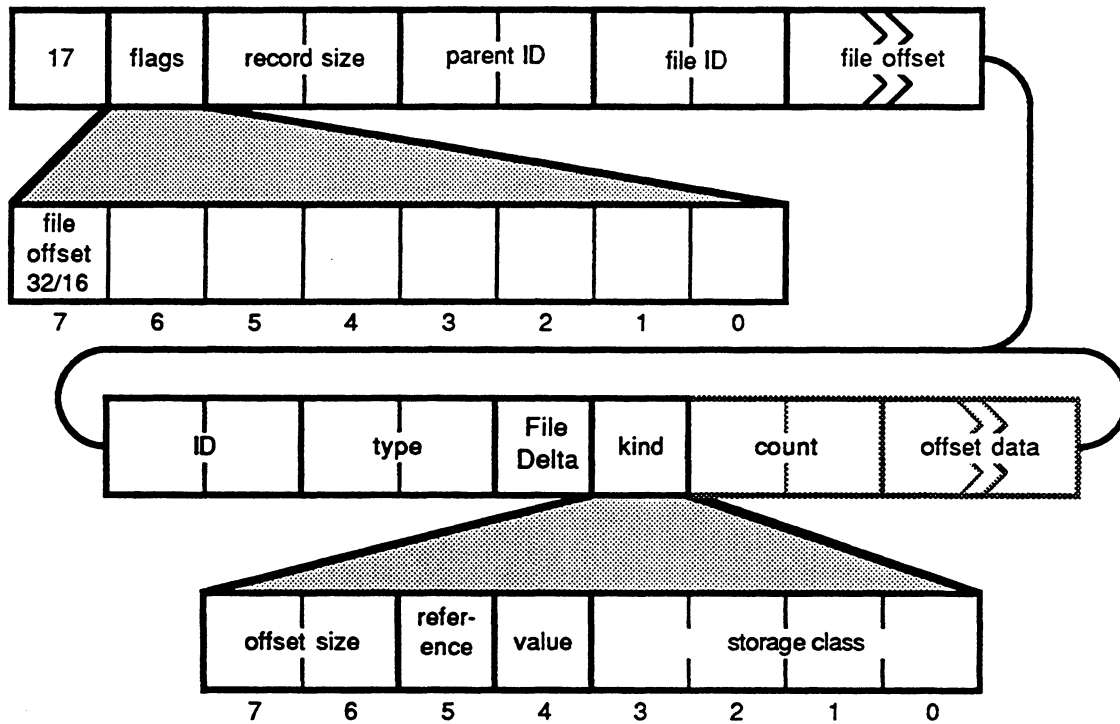
The record size field is used to determine the number of IDs which follow.

The parent ID specifies the scoping entity (such as module or block) containing the identifier.

The file ID associates a dictionary ID with a filename.

The file offset specifies the source file offset for the first statement specified by this Local Identifier Record. This field may be either a 16- or 32-bit signed value giving the zero-relative byte offset in the file specified by the file ID.

Following the initial file offset one or more sets of 5 fields will provide information about local identifiers.

The ID associates a dictionary ID with a formal parameter or a local variable.

The type is a type ID specifying the identifier's type. For primitive types, this ID will range from 0 to 99; for nonprimitive types it will be the ID (≥100) of the LocalType record describing the type.

The File Delta byte specifies the identifier's source offset. (The first delta byte will usually be zero.) If the delta for the variable is not in the range 0..255 then a new record must be started.

The Kind byte is used to determine whether the identifier is in fact a formal parameter, a function return "parameter," or a local variable. The Kind field also specifies the size of the byte offset which follows, as well as the storage mode for the parameter or variable.

The offset size field indicates whether there is a byte offset field present, and if there is, how large it is. If it is three, a 16-bit count field specifies the number of bytes that follow the count (used for representing floating-point or string constants). If the count is odd, a pad byte should be added to the data.

offset size	description
0	no offset field follows
1	2-byte offset field
2	4-byte offset field
3	variable size offset field (preceded by 16-bit count)

The reference and value bits in the Kind field must be 0 in the case of local identifiers, and must be specified in the case of formal parameters.

storage class	description	offset is the...
0	register	register number
1	A5-relative	ID of the module or entry point corresponding to this variable
2	A6-relative	A6-relative offset
3	A7-relative	A7-relative offset
4	absolute	absolute address
5	constant	value of the constant
6...15	*reserved*	*reserved for future use*

A7-offsets (storage class 3) are encoded as offsets from the top of the stack (usually the return address) *prior* to executing a LINK instruction.

Register numbers are encoded as integers indicating the specific register as follows:

■ **Table H-1**   Register numbers

Value	Register	Meaning
0..7	D0..D7	Data registers
8..15	A0..A7	Address registers
16	CCR	Condition code register
17	SR	Status register
18	USP	User stack pointer
19	MSP	Master stack pointer
20	SFC	Source function code register
21	DFC	Destination function code register
22	CACR	Cache control register
23	VBR	Vector base register
24	CAAR	Cache address register
25	ISP	Interrupt stack pointer
26	PC	Program counter
27		*reserved*
28	FPCR	Floating-point control register
29	FPSR	Floating-point status register
30	FPIAR	Floating-point instruction address register
31		*reserved*
32..39	FP0..FP7	Floating-point data registers
40..50		*reserved*
51	PSR	PMMU status register
52	PCSR	PMMU cache status register
53	VAL	PMMU validate access level register
54	CRP	PMMU CPU root pointer register
55	SRP	PMMU supervisor root pointer register
56	DRP	PMMU DMA root pointer register
57	TC	PMMU translation control register
58	AC	PMMU access control register
59	SCC	PMMU stack change control register
60	CAL	PMMU current access level register
61..62	TT0..TT1	MC68030 transparent translation registers
63		*reserved*
64..71	BAD0..BAD7	PMMU breakpoint acknowledge data registers
72..79	BAC0..BAC7	PMMU breakpoint acknowledge control registers

Local Label record

Local Label records give the correspondence between generated code, source statements, and label identifiers.
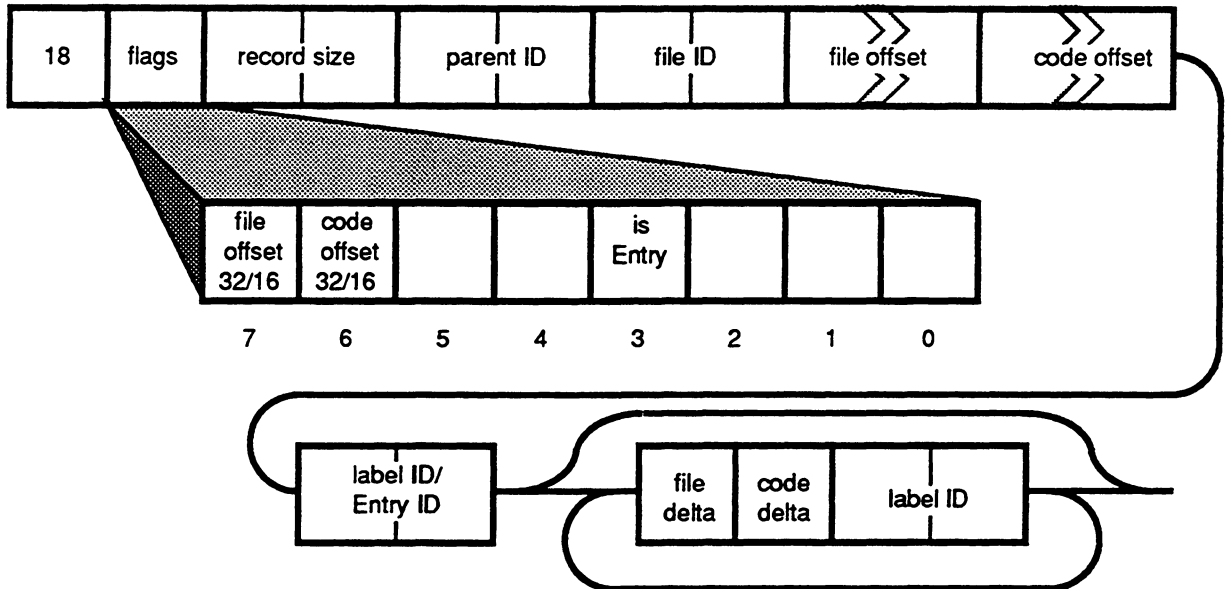
These records are similar to Source Statement records.

The flags, record size, parent ID, file ID, file offset, and code offset fields are identical to those fields in the Source Statement record (see description earlier in this appendix).

The file delta and code delta are also encoded as they are in Source Statement records, but here an additional label ID field is supplied with each file and code delta pair. The label ID associates a dictionary ID with a label identifier. The *first* label ID in the Local Label record has no file and code delta following it since that ID is at the location specified by the file and code offset fields.

Local Type record

Local Type records associate type declarations with type IDs (or several types with several IDs). Type IDs are used to define types referred to by LocalID and other Local Type records.
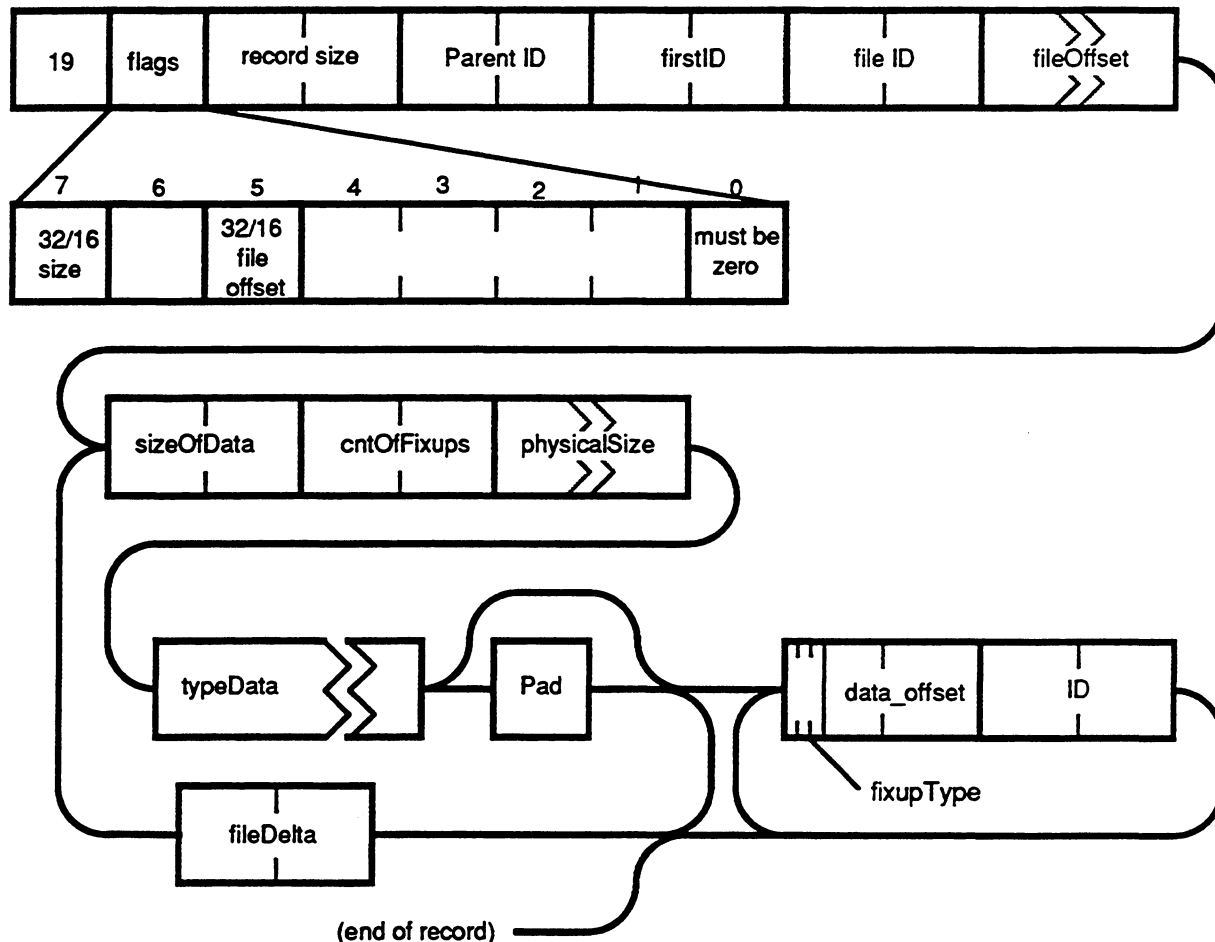
The record size field specifies the total number of bytes in the record.

The parent ID specifies the scoping entity (such as a module or block) containing the identifier.

The 32/16 size bit in the flags field specifies whether the physicalSize fields of all type declarations in the record are 16 or 32 bits.

The 32/16 file offset bit in the flags field specifies whether the fileOffset field is 16 or 32 bits.

Bit zero of the flags field must be zero.

The firstID field specifies the first type ID declared by the record, 'firstID + 1' is associated with the second type declared, and so on. The first type ID should be ≥ 100; IDs 0 through 99 are treated as "primitive" types and should never be defined by a LocalType record.

The fileID field specifies the dictionary ID containing the source filename. If the file ID is 0 then the type declarations in the record are not associated with any source code, and the fileOffset and fileDelta fields are ignored (though they must appear).

The fileOffset field specifies the offset of the type declaration in the source file. This is a 16-bit or 32-bit unsigned value, as determined by the 32/16 file offset bit in the flags field.

Type declarations immediately follow the firstID field, and extend to the end of the record. For each declaration the following fields appear:

- The sizeOfData field specifies the size of the type data in bytes. (If this field is odd, a pad byte of zero must follow the type data.)

- The cntOfFixups field specifies the number of type ID and dictionary ID fixups that follow the type data.

- The physicalSize field is either 16 or 32 bits (see definition of the flags field) and specifies the type's physical representation size, in bytes (such as the value returned by C's sizeof operator). This field may be inaccurate for packed types and sliced arrays, and is meaningless for ProcOf types.

- The type data follows the physicalSize record. The format of the type data is described in the following section, "Type Interpretation via Prefix Code." If the size of the type data is odd then a pad byte of zero must be appended to it. The pad byte is not included in the size of the type data.

- Following the type data is the fixup list, consisting of cntOfFixups entries. Each fixup entry is two words: an offset into the original type data followed by an ID to translate. The ID is translated and the result is inserted into the type data at the specified offset. The upper three bits of the fixup offset indicate the kind of translation to be made; the remaining bits are the type data offset. The translation kinds are:

bit 15	bit 14	bit 13	Translation treatment
0	0	0	Insert the TTE index of the ID (which must be a LocalType or a number from 0 to 99) as a 16-bit word
0	0	1	*reserved*
0	1	0	Insert a scalar based on the ID's type:

Module   insert MTE index of the module
LocalID   insert CVTE index of the variable
Segment   insert RTE index of the segment
Source file   insert FRTE index of the source file
LocalType   insert TTE index of the type

bit 15	bit 14	bit 13	Translation treatment
0	1	1	*reserved*
1	0	0	Insert NTE index of the ID, as a scalar
1	0	1	*reserved*
1	1	0	*reserved*
1	1	1	*reserved*

(MTEs, CVTEs, RTEs, FRTEs, and TTEs and so forth are part of the Sym file produced by the linker, and are described in the document *SADE Sym File Format,* which is available separately from Developer Technical Support at Apple Computer, Inc.)

■ The fileDelta field indicates the source file offset of the next type declaration as a *signed* 16-bit integer. This field does not appear following the last type declaration in the record. If the source offset will not fit in 16 bits, then a new Local Type record should be started.

A type ID is associated with a name by a "DbgId"-type dictionary entry with a corresponding ID.

A new Local Type record should be started whenever the source file changes, when the file delta cannot be represented in a 16-bit signed integer, or when a type has no source declaration.

Type interpretation via prefix code

The goals of type interpretation are to support the interpretation of SADE debugger variables by type, map from name to type, and map from type information to name. Operators such as array indexing, indirection, field name, and so on, are applied to types to yield more type and address information.

The type information should be complete. By *complete* is meant that the debugger should have a minimum amount of knowledge about how data for any particular type is stored. Type information should be easy for compilers to emit and for the linker to extract from the Object module format. The storage of type information should be compact, yet fast and easy to expand. High-level language interpreters for type information should not be prohibitively expensive.

Overview

SADE type information is contained in word aligned variable length data structures called *Type Table Entries, (TTE)*. These type table entries are contiguously numbered (starting at 100, as the indices 0 .. 99 are reserved) and accessed via an indexing table of four byte disk addresses. An index into this table is called a *TTE Index*. The type table entries are aligned on word (two byte) boundaries. No global/local scope information is contained in the type table; scoping is via the Modules table and its Contained Modules, Types, and so on. A picture of a *TTE* appears later in this appendix.

The TypeCodes portion of a TTE contains size information and an interpretative representation of a type. The exact form for the TypeCodes is described below. The paradigm used for the SADE type mechanism is types as functions. All types are either basic types or functions with types and integral constants as arguments. The realization of this paradigm is prefix code: an operator followed by operands.

Type values are either scalar types or composite types. Instances of a scalar type can be ordered, while composite types cannot necessarily be ordered. Scalar types are further divided into integral and nonintegral scalar types. Integral types can be mapped to the set of integers.

Type functions

Following are the definitions for the type functions and their arguments. Except for two cases, type function arguments are either other type functions, encoded scalars, or instances of a scalar type (see the section "ScalarOf"). In one exception, *ConstantOf( )*, one argument is a sequence of uninterpreted bytes. In the other, *TTE( )*, the argument is an unaligned 2 bytes interpreted as an unsigned word. Except in these two cases, the convention used is to prefix arguments with S if it is a scalar or T if it is a type or an instance of a type.

BasicType(SType)

This function returns a ground type, one which cannot be composed of other types[†]. The argument is an integer in the range 0–99. By convention, the empty type, called *void*, is represented by BasicType(0).

TTE(UnsignedWord)

The type, and also the name for the type, is found at the UnsignedWord entry in the type table. This aliasing function allows the association of a name to a type, or the factoring-out of a shared anonymous type into one common entry, thus saving table space.

PointerTo(Ttype)

The argument is a type. The value of the function is pointer to that type. PointerTo(void) is a generic pointer, equivalent to the C type (void *).

ScalarOf(Ttype, Svalue)

The type returned by the ScalarOf function names an instance of given scalar type. The ScalarOf function is usually referred to by RecordOf or EnumerationOf.

NamedTypeOf(Snte, Ttype)

The scalar is an index for a Name Table Entry. That entry gives the name for the type Ttype. This mechanism is used to give names which are local to a type, such as record and union field names.

ConstantOf(Ttype, Slength, byte...)

The function ConstantOf is similar to ScalarOf, except that the constant can be of a nonintegral type, such as floating-point constants or composite type constants. The TType is the type of the constant, the Slength is the number of bytes in the constant, and the unencoded bytes following are the bytes comprising that type.

[†] This is not quite true. Due to historical reasons, strings are considered a ground type instead of a derived type.

EnumerationOf(Tbase, Slower, Supper, Snelements, Ttype...)

The function EnumerationOf names an enumeration type. Tbase names the underlying scalar type that the elements of the enumeration are drawn from. It also determines the storage size of the Enumeration. Usually, the elements are drawn from BasicType(SignedWord). The Slower and Supper are the lower and upper bounds of the enumeration. Snelements is the number of Ttype elements named as part of the enumeration. Snelements can be less than Supper Slower +1 if the enumeration is sparse, as is possible with C enums.

VectorOf(Tindex, Telement)

The function takes two arguments. The first is the index type, which is the scalar type from which the vector indices are drawn. The second argument is the type of the vector elements. The value of the function is Array [Tindex] OF Telement.

RecordOf(Snfields, Soffset & Ttype...)

The function RecordOf returns a type composed of a linear sequence of types. Snfields is the number of types in the composite type. The argument types are pairs of a scalar and a type. The Soffset scalar gives the offset to that element from the beginning of the type. The Ttype is the type of that element. The offsets are byte offsets. The representation details are discussed in the following section.

UnionOf(Ttag, Soffset, Snfields, Tvariant & Ttype...)

The Union function could be built from the RecordOf function. Ttag is the scalar type of the tag. For C and other languages whose unions do not have a tag, Ttag is the ground type void. Soffset is the offset to the first variant from the start of the union. If there is no tag variable, then Soffset will be 0. Otherwise, Soffset is the size of the tag variable plus any required alignment. The Snfields is the number of variants in the union. The Tvariant and Ttype pairs define one element of the union. The Tvariant is an instance of the Ttag and names the variant. If Ttag is void, Tvariant is void. Ttype is the field of the union.

SubRangeOf(Tbase, Tlower, Tupper)

The function SubRangeOf names a subrange of the scalar type, Tbase. The bounds of the subrange are given by the Tlower and Tupper values.

SetOf(Tbase)

The function SetOf names a set type. The set is composed of elements drawn from scalar type Tbase.

ProcOf(SClass, TReturn, SArgc, TArg...)

The function ProcOf names a procedure type. SClass is the *class* of procedure. The class of the procedure defines how arguments are passed to it and values are returned from it. TReturn is the type of the return value; if *void* the ProcOf names a procedure instead of a function. SArgc is the number of arguments to the procedure. The TArg are the arguments to the procedure. The argument order is the order of declaration, as it appears in the source text. How the TArgs are actually passed, on the stack, Pascal or C, or in registers, is as per the SClass.

ValueOf(Ttype, Scvte, Smte)

The scalar value, of type Ttype, can be obtained by fetching the variable whose CVTE index is Scvte and whose containing module's MTE index is given by Smte. Smte is required because the variable named by the Scvte might be in a register or relative to A6, requiring a debugger to find the proper stack frame. Ttype will be the same as the type in the CVTE; it is duplicated because it's not desirable to reference the CVTE just to find the type.

ArrayOf(Telement, Sorder, Sndirm, Tbound1, ...)

The ArrayOf type descriptor is used to describe monolithic arrays, those which cannot or should not be described with VectorOf functions. The Telement is the type of each array element. Sorder describes how the address of an element is computed from the array indices. Sndim is the number of dimensions of the array, and Tbound1... are the indexing types, usually SubRangeOf types.

Sorder can have one of two values. If 0, then the address of array(i,j,k) is computed by

(i-*lowerBound*(Tbound1)) +
(j-*lowerBound*(Tbound2)) * *span*(Tbound1) +
(k-*lowerBound*(Tbound3)) * *span*(Tbound2) * *span*(Tbound1)

If 1, then the address is computed by

(i-*lowerBound*(Tbound1)) * *span*(Tbound2) * *span*(Tbound3) +
(j-*lowerBound*(Tbound2)) * *span*(Tbound1) +
(k-*lowerBound*(Tbound3))

where

lowerBound(Type)	= lower bound of the subrange or enumeration
upperBound(Type)	= upper bound of the subrange or enumeration
span(Type)	= *upperBound*(Type) - *lowerBound*(Type)

Note the difference in the order of the element type and index type from that in the Vectorof function. This was done to keep the variable number of bounds as the last arguments to the Arrayof function.

Representation of type information in the SADE symbol table

The SADE symbol table, created by the linker from information in the object modules linked, stores type information in a type table Entry. The type table Entry is a word-aligned variable-length data structure of the form:
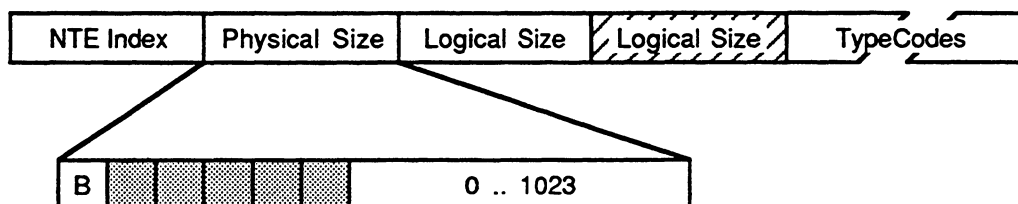
NTE index

The NTE index is a 4-byte field, giving the name of the type via an index into the Name Table. The index can be zero, meaning that this is an anonymous type.

Physical Size

The Physical Size 2-byte field defines the number of bytes taken up by the TypeCodes field. It does not include the physical size or logical size fields. The maximum size of a TypeCodes field is 1023 bytes. The other bits are flag bits. Of these flag bits, only one, the *B* bit, is defined. The other bits are reserved for future expediencies and should be set to zero. The B bit applies to the Logical Size and means *Big*. When set, the logical size field will be 4 bytes instead of 2, allowing the description of very large data structures such as Fortran arrays.

Logical Size

The Logical Size field is either 2 bytes or 4 bytes wide. If the B bit in the Physical Size field is zero, then the Llogical Size is 2 bytes long. If set, the field is 4 bytes long. In either case, the value of the field is the number of bytes required to store the type.

TypeCodes

Following the header information are the TypeCodes themselves. This is a sequence of *Physical Size* bytes of type information. The TypeCodes are described in the next section.

Representation of type codes

Type function codes are single bytes. The Basic Type function has the Most Significant Bit (MSB) of the byte 0 and the basic type number encoded in the lower 7 bits. The Type Composition Function has a 1 in the MSB and the function code in the lower 6 bits. The next-to-MSB is a flag indicating the type of offset encountered in the arguments to that type. If zero, then offsets are *byte* offsets from the beginning of the type. If 1, then offsets are *bit* offsets from the beginning of the type.

Basic type

```
┌───┬──────────────────────┐
│ 0 │       0 .. 99        │
└───┴──────────────────────┘
```
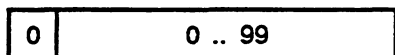
The byte names a Basic type in the range 0 .. 99. The standard values for Basic types are named earlier in this appendix and are as follows:

0	no type
1	Pascal string
2	unsigned long word
3	signed long word
4	extended (10 bytes)
5	Pascal boolean (1 byte)
6	unsigned byte
7	signed byte
8	character (1 byte)
9	character (2 bytes)
10	unsigned word
11	signed word
12	singled
13	double
14	extended (12 bytes)
15	computational (8 bytes)
16	C string
17	as-is string

Type composition function

```
+---+---+-------------+
| 1 | P |   0 .. 63   |
+---+---+-------------+
```
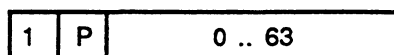
The type code is in the lower 6 bits. The P bit means the type is PACKED. The default, P=0, means that the type is unpacked. The values for the type codes are:

1	TTE
2	PointerTo
3	ScalarOf
4	ConstantOf
5	EnumerationOf
6	VectorOf
7	RecordOf
8	UnionOf
9	SubRangeOf
10	SetOf
11	NamedTypeOf
12	ProcOf
13	ValueOf
14	ArrayOf

The SClass for ProcOf types has the following immediate meanings:

0 Undefined

The calling conventions of the function are undefined.

1 Pascal

The calling convention follows Pascal conventions. The return value or a pointer to the return value is placed on the stack and is then followed by a fixed number of arguments. The called procedure/function cleans up the stack before returning to the caller.

2 C with fixed arguments

C calling convention. The caller removes any arguments that were passed. The arg count is the number that was specified in the prototype for the C function or in the function definition.

3 C with variable number of arguments

Similar to the above, except that the argument count is the minimum number of arguments to pass to the function. It is suggested that compilers emit a ProcOf(3, TReturn, 0) for functions referenced without prototypes or function definitions, as that is the most general case. This is especially useful for declarations of pointers to function.

Representation of scalars

0	0 .. 127
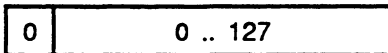
A constant in the range 0–127 fits into a single byte. The high bit is zero, marking this as a small integer.

1	1	-63 .. -1
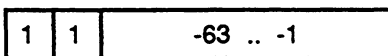
If the 2 high bits of a byte are one, then the byte represents a small negative integer in the range –63 –1. The value –64, with the lower 6 bits zero, is used as switch to long.
See below.
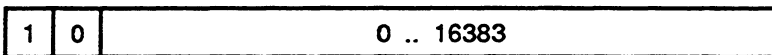
1	0	0 .. 16383

If the two high bits of a byte are 10, then a larger integer in the range 0 .. 16383 is specified by appending the next byte to the 10 byte and stripping off the high two bits of the word. The constant is not aligned to word boundaries.
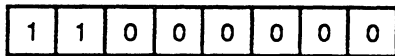
```
┌───┬───┬───┬───┬───┬───┬───┬───┐
│ 1 │ 1 │ 0 │ 0 │ 0 │ 0 │ 0 │ 0 │     4 bytes of longint
└───┴───┴───┴───┴───┴───┴───┴───┘
```

A byte whose value is –64 is used for expansion past the preceding bounds. The four bytes
following (not aligned to word boundaries) are the `longint`.

Examples

In these examples, it is assumed that "Integer" is the name of a BasicType of SignedWord.

Pascal source

```
TYPE
        VHSelect    = (v, h);

        Point       =
          RECORD
              CASE Integer OF
                 0: (v: Integer;
                     h: Integer);
                 1: (vh: ARRAY [VHSelect] OF Integer);
          END;

        Rect        =
          RECORD
              CASE Quux: Integer OF
                 0:(top:     Integer;
                     left:        Integer;
                     bottom:      Integer;
                     right:       Integer);
                 1:(topLeft:      Point;
                     botRight:    Point);
          END;
```

Possible object module representation

The representation of type information in the object module must be less compact than
in the type table so that the linker will not have to interpret the type information, just
resolve linkages, fold anonymous types into types referring to them, and emit the
compacted information. The differences are:

- TTE(x) are replaced by the dictionary numbers of the entries.
- BasicType(x) are replaced by dictionary numbers less than 100.

In the following, dictionary numbers are distinguished from integers. BasicType(*n*) should be viewed as having been replaced by the canonical dictionary for that type.

Dict #	Name	Type definition
3000	v	ScalarOf(3020, 0)
3010	h	ScalarOf(3020, 1)
3020	VHSelect	EnumerationOf(BasicType(0), 0, 1, 2, 3000, 3010)
3050	vh	VectorOf(3020, BasicType(INTEGER))
3052		ScalarOf(BasicType(INTEGER), 0)

```
3054            RecordOf(
                    /* Record has two fields */
                    2,
                    /* First, at offset 0, is the "v: Integer"*/
                    /* Two bytes after it is "h: Integer"*/
                    0, NamedTypeOf("v", BasicType(INTEGER)),
                    2, NamedTypeOf("h", BasicType(INTEGER))
                    ),
3056            ScalarOf(BasicType(INTEGER), 1)
```

Dict #	Name	Type definition
3060	Point	UnionOf(

```
                    /* Two variants, Integer selector, no tag */
                    2, BasicType(INTEGER),
                    3052, 3054,       /* 0: selects the record of
                            "v" and "h" */
                    3056, 3050        /* 1: selects the array of
                            two ints */
                    )
3132            RecordOf(
                    4,
                    0, NamedTypeOf("top", BasicType(INTEGER)),
                    2, NamedTypeOf("left", BasicType(INTEGER)),
                    4, NamedTypeOf("bottom", BasicType(INTEGER)),
                    6, NamedTypeOf("right", BasicType(INTEGER))
                    )
3134            RecordOf(
                    2,
                    0, NamedTypeOf("topLeft", 3060),
                    4, NamedTypeOf("botRight", 3060)
                    )
```

Dict #	Name	Type definition
3140	Rect	UnionOf(

```
                    /* Two variants, Integer selector,
                            tag is "Quux", size 2 */
                    2, NamedTypeOf("Quux", BasicType(INTEGER)), 2
                    3052, 3132,       /* 0: First variant is the
                            record of 4 Integers */
                    3056, 3134        /* 1: Second variant is the
                            record of 2 Points */
                    )
```

Possible compilation into TTE

The linker can compact the representation, folding anonymous entries into the type definitions referring to them. This would yield the following:

```
3100            v           ScalarOf(TTE(3102), 0)
3101            h           ScalarOf(TTE(3102), 1)
3102        VHSelect        EnumerationOf(BasicType(SIGNEDWORD), 0, 1, 2, TTE(3100),
                            TTE(3101))
3105            vh          VectorOf(TTE(3102), BasicType(INTEGER))
3106        Point           UnionOf(
                                    /* Two variants, Integer selector, no tag */
                                    2, BasicType(INTEGER),

                                    /* 0: selects the record of "v" and "h" */
                                    ScalarOf(BasicType(INTEGER), 0),
                                    RecordOf(
                                            /* Record has two fields */
                                            2,

                                            /* First, at offset 0, is the
                                                    "v: Integer" */
                                            0, NamedTypeOf("v", BasicType(INTEGER)),

                                            /* Two bytes past the first, is
                                                    "h: Integer" */
                                            2, NamedTypeOf("h", BasicType(INTEGER))
                                    ),

                                    /* 1: selects the array of two ints */
                                    ScalarOf(BasicType(INTEGER), 1),
                                    TTE(3105)
                            )
```

```
3114          Rect          UnionOf(

                            /* Two variants, Integer selector,
                                    tag is "Quux", size 2*/
                            2, NamedTypeOf("Quux", BasicType(INTEGER)), 2

                            /* 0: First variant is the record
                                    of 4 Integers */
                            ScalarOf(BasicType(INTEGER), 0),
                            RecordOf(
                                    4,
                                    0, NamedTypeOf("top", BasicType(INTEGER)),
                                    2, NamedTypeOf("left", BasicType(INTEGER)),
                                    4, NamedTypeOf("bottom", BasicType(INTEGER)),
                                    6, NamedTypeOf("right", BasicType(INTEGER))
                                    ),

                            /* 1: Second variant is the record
                                    of 2 Points */
                            ScalarOf(BasicType("Integer"), 1),
                            RecordOf(
                                    2,
                                    0, NamedTypeOf("topLeft", 3106),
                                    4, NamedTypeOf("botRight", 3106)
                                    )
                            )
```

Type interpretation and packed data

The previous section did not describe PACKED data, such as Pascal's packed arrays (actually vectors) and records and C's bit-field structures. This proposal for describing packed data rests on the observation that it is not the composite types that are packed but rather components of the composite types. The solution is general enough to solve today's problem and be extensible to future language implementations.

When a type has the PACKED attribute, it is followed by packing information. The packing information describes the packing for that occurrence of the type. In the case of the *VectorOf* composing type, the packing information describes how a given number of elements are arranged within a fixed number of bytes.

Storage framework

In general, an instance of a datatype is a sequence of bits. These bit sequences are usually aligned on address boundaries and are totally contained within an integral unit of addressable units of storage (bytes). Packing an instance of a datatype minimizes ·the amount of wasted bits, possibly at the expense of access time. Therefore, the packed type information assumes the following bit-level view of **K** bytes of storage, with the most significant bit (and byte) at the left and the least significant bit (byte) at the right:
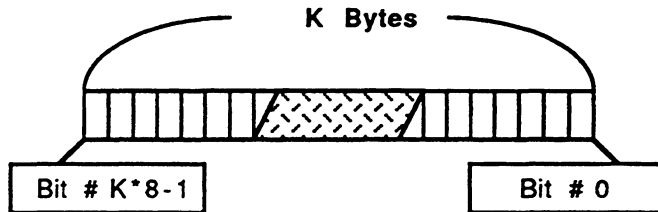
A packed data field is represented by two scalars: the **MSB** and **LSB**. The scalars are represented as described in the TypeCodes document. This allows bitfields and packed data to be from 1 to $2^{**}32-1$ bits wide[†]. The number of bytes occupied by that packed data field is (MSB+7) DIV 8. LSB can be thought of as the shift factor and MSB–LSB+1 the number of bits in the mask.

A packed non-*VectorOf* type is followed by a single occurrence of MSB and LSB.

Packed vectors have a regular structure. A group of vector elements, M, will be packed into a number of bytes, N. The packing is regular, meaning that the bit field for Vector[p*M+k] is the same as that for Vector[k] except that it is p*N bytes after Vector[k]. Because of this regularity, the bitfield information is in a different format:

[*†] Originally, it was thought that the packed data field could be represented by a single byte of mask and shift information. One nibble in the byte represented the field width (0 to 15 bits) in the word. The position of the field was represented by the other nibble as the number of bits to left shift the mask. However, one common case, the C bitfield with more than 15 bits, prevents a single byte representation for packed data. Since more than one byte is required to represent packed data, the MSB/LSB view was adopted.

MSB/LSB was chosen over MASK and SHIFT because one less addition is required to yield the number of bytes required for the data type.

The boxes are labeled (vertically): # Bytes/Group, # Bits/Element, # Elements/Group, # LSB [0], # LSB [M-1]
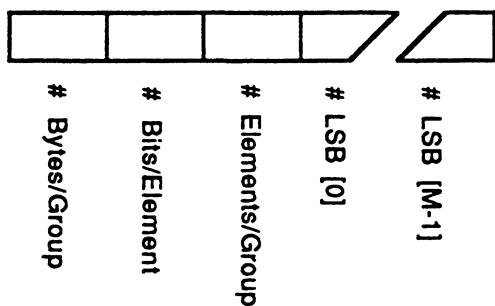
The first scalar gives N, the number of bytes into which the M vector elements can fit. The next scalar gives the width of the bit mask. The third scalar gives M, the number of elements in the repeating group. Following these three scalars are M scalars. These M scalars give the LSB for the 0'th to the M-1'th vector element. LSB[i]+# Bits/Elements-1 gives MSB[i].

Examples

The following example is in C and from Harbison and Steele. It is assumed to follow MC68000 addressing in that the most significant bytes are at the lowest addresses. Also note that the example assumes that this particular C compiler packs fields into `int`s starting with the least significant bit of the `int`.

Here is a sample bitfield:

S	x	Segment	Page	Offset

31 29   23   15   0
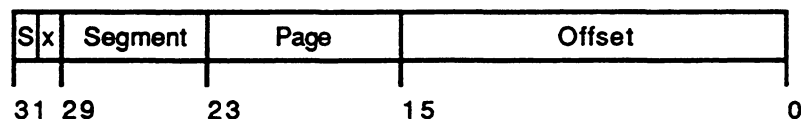
C source

```
typedef struct {
        unsigned offset    : 16;
        unsigned page         : 8;
        unsigned segment   : 6;
        unsigned           : 1;   /* For future use */
        unsigned supervisor   : 1;
} V_ADDR;
```

Possible compilation into TTE

The preceding fields are not all packed fields. Only the segment and supervisor fields need to have the PACKED attribute applied to them. Those two fields are followed by the bit positions within the byte.

```
3107  V_ADDR      RecordOf(
                     4,

                     /* First is "offset".  Note its byte offset */
                     2, NamedTypeOf("offset", «UnsignedInteger»),

                     /* Next is "page".It is physically before "offset" ⌐
                     1, NamedTypeOf("page", «UnsignedInteger»),

                     /* Finally, "Segment" and "Supervisor" in byte 0*/
                     0, PACKED NamedTypeOf("segment",
              «UnsignedInteger»), 5, 0,
                     0, PACKED NamedTypeOf("supervisor",
              «UnsignedInteger»), 7, 7
                     )
```

Appendix I **In Case of Emergency**

THIS APPENDIX CONTAINS SOME INFORMATION THAT MAY BE USEFUL
when serious system errors occur. ■

Contents

Crashes

If you end up in the MacsBug debugger while running MPW, it may be possible to recover without rebooting and losing your recent changes. Type G SYSRECOVER. The Shell will attempt to recover by aborting the current command, saving the contents of all the windows, and/or returning to the Finder. If this fails, type ES to return to the Finder, then shut down the system immediately.

Stack space

The MPW Shell and tools that run integrated with the Shell share a single stack. The stack size is determined by the Shell at initialization time. Complex command files, large links, and other tools may require more stack space than is available. System errors 28, 2, and 3 are possible indications of this problem. You can increase the stack size by using ResEdit to modify 'HEXA' resource number 128 in the file MPW Shell. The default size is $2710 (10,000 bytes) when less than 480,000 bytes are available for the application heap and $4E20 (20,000 bytes) when more than 480K are available.

Glossary

abstract target: In Make dependency rules, a target that is not actually built but represents a collection of items. Use an abstract target to trigger dependencies on the right side of the dependency rule. See Chapter 9 for details on Make dependency rules.

active window: The frontmost window. The Shell variable {Active} always contains the name of the current frontmost window.

alias: An alternate name for a command, defined with the Alias command.

application: A program that runs stand-alone, outside of the Shell environment. An application's file type is APPL.

author: In Projector, with respect to a particular revision, the name of the person who made a revision. With respect to Projector's files and projects, the person with the primary responsibility for a file or project.

blank: A space or a tab character (in the context of separating words in the command language).

branch: In Projector, an additional sequence of revisions emanating from another revision and running parallel to the main trunk.

build commands: Shell commands that are output by the Make tool, used to build a program.

build command line: In the dependency rule of a makefile, the lines beginning with a space or tab that follow the dependency line.

built-in commands: Editing commands, structured commands, and other Shell commands that are part of the MPW Shell application (as opposed to *MPW tools*, which are separate files on the disk.)

CheckOut directory: The directory into which, by default, Projector places checked out files. Each project has a corresponding CheckOut directory which can be changed with the CheckOutDir command.

'ckid' resource: The "check ID" resource that Projector maintains in the resource fork of all files belonging to a project. The 'ckid' resource contains identification such as User name, Task, Project, and so on.

code resource: A resource that contains a program's code—most commonly a resource of type 'CODE' (for applications and MPW tools) but also other resource types such as 'DRVR' and 'PDEF' that also contain code.

command file: See **script.**

command name: The first word of a command, identifying the name of a built-in command or the name of a file (tool, command file, or application) to execute.

command script: See **script.**

command substitution: The replacement of a command by its output. Command substitution takes place within back quotes (`...`).

comment: In Projector, user-supplied text describing a particular revision, file, or project.

console: The window where a command is entered and executed (standard input). Also, the window to which the command's output is returned (standard output).

current project: In Projector, the name of the current project. Projector assumes all actions pertain to this project unless a different project is specified with the –project option.

current selection: The currently selected text in a window. In editing commands, the current selection in the target window is represented by the § metacharacter.

data fork: The part of a file that contains data accessed via the Macintosh File Manager.

data initialization interpreter: The module _DATAINIT in the libraries Runtime.o and CRuntime.o.

dead code: In the linker, modules that cannot be reached from any references available, given a main module. See "Dead Code" in Chapter 8 for more information.

dependency file: A makefile.

dependency line: In Make, the first line of a dependency rule. See also: **dependency rule.**

dependency rule: In Make, a rule that specifies the prerequisite files of a given target file, along with a list of the commands needed to build the target file.

dependent: In a Commando dialog, a control that is enabled or disabled depending on the state of its parent control.

derez: To decompile a resource file by using the MPW resouce decompiler, DeRez.

desk accessory: A "mini-application," implemented as a device driver, that can be run at the same time as an application. Desk accessories are files of type DFIL and creator DMOV, and are installed by using the Font/DA Mover.

device driver: A program that controls the exchange of information between an application and a device.

diagnostic output: Commands and tools send error and progress output to diagnostic output (by default, the window where the command was executed). You can redirect diagnostic output to another file, window, or selection with the ≥ and ≥≥ operators. Diagnostic output is also referred to as "standard error."

dialog: In Commando, the programmed interaction between a user and a tool or script. A dialog may utilize more than one **dialog box.**

dialog box: A window that appears when a command is invoked, offering options and parameters.

Directed Acyclic Graph: The MPW linker creates a tree of all reachable modules from a given main module. See "Dead Code" in Chapter 8 for more information.

Editor: When appearing with initial capitals, the built-in commands appearing in MPW's Edit menu, a part of the MPW Shell.

entry point: A location (offset) within a **module.**

escape character: The Shell escape character is ∂ (Option-D). It is used to disable (or "escape") the special meaning of the character following it, to continue commands over more than one line (∂Return), and to insert invisible characters into command text.

external: A module, entry point, or segment that can be referenced from different object files.

external reference: A reference to a routine or variable defined in a separate compilation or assembly.

file information: Information maintained by Projector on a per-file basis. Examples are: Author, Last Modification Date, and Comment.

filename: A sequence of up to 31 printing characters (excluding colons), that identifies a file. See also **pathname.**

file type: A four-character sequence, specified when a file is created, that identifies the type of file. (Examples: TEXT, APPL, MPST.)

Finder information: Information that the Finder provides to an application upon starting it, telling it which documents to open or print.

Font/DA Mover: An application, available on the *System Tools* disk, used for installing desk accessories in the System file.

full pathname: See **pathname.**

HFS: Hierarchical File System used on 800K disks and the Apple hard disks.

ID: A file-relative number for identifying a **module,** an **entry point,** or a **segment,** within a single object file.

insertion point: An empty selection range; that is, the character position where text will be inserted (marked with a blinking vertical bar).

Integrated Environment: A set of routines, modeled on the C language, that provide parameter passing, access to variables, and other functions to MPW tools. (See Chapter 12 and Appendix F.)

interface routine: A routine called from C, Pascal, or Assembler whose purpose is to trap to a certain ROM or library routine.

jump table: A table that contains one entry for every routine in an application or MPW tool, and is the means by which the loading and unloading of segments is implemented.

leafname: A partial pathname that contains no colons. A leafname might be a directory and a filename, such as "Tools:ResDet" or simply a filename. MPW assumes the default directory. See also: **pathname** and **partial pathname.**

literal: In the resource compiler, Rez, a value within single quotation marks. (See Chapter 8.)

local: A **module, entry point,** or **segment** that can be referenced only from within the file where it is defined.

location map: The linker can write to standard output a map of memory segments sorted by *segNum* and *segOffset.* (See Chapter 10.)

locked revision: In Projector, a revision currently checked out for modification.

main segment: The segment containing the main program or procedure.

makefile: Used by the Make command, a file that describes dependencies between the various pieces of a program, and contains a set of commands for building up-to-date files. The default makefile is named MakeFile.

module: A contiguous region of memory that contains code or static data. A module is the smallest unit of memory that is included or removed by the linker.

mounted project: In Projector, a project that is not nested within another project. Similar to the root directory on a volume. You can mount several projects, just as you can mount several volumes. You can access all projects under the mounted project.

MPW Shell: The application that provides the environment within which the other parts of the Macintosh Programmer's Workshop operate. The Shell combines an editor, command interpreter, and built-in commands.

MPW tool: An executable program (type MPST) that is integrated with the MPW Shell environment (contrasted with an application, which runs stand-alone). Like applications, tools exist as separate programs on the disk.

name: In Projector, an identifier that represents a set of files, revisions, and branches, with the restriction that a name can refer to only one revision in any one file.

non-HFS: The nonhierarchical file system, used on 400K disks and Macintosh XL hard disks.

option: A command-line switch, specifying some variation from a command's default behavior. Options always begin with a hyphen (-).

orphaned file: In Projector, a file that belongs to a project, but whose resource fork no longer contains the information that Projector needs to determine to which project it belongs.

parameter: The words following the keyword in a simple command. There are two types of parameters: **options** and **files**. Note that certain parameters, such as I/O redirection, are interpreted by the Shell and never seen by the command itself.

parent: In Commando, an option or control whose status determines whether a dependent option or control is enabled or disabled.

partial pathname: A pathname that either contains no colons or has a leading colon. See also: **leafname**.

pathname: A sequence of up to 255 characters that identifies a file, directory, and/or volume. A *full pathname* contains embedded colons but no leading colon. It specifies *volume: directory:...filename*. A *partial pathname* either contains no colons or has a leading colon. A partial pathname is convenient to use if the file is located in the current default directory. A *leafname* is a partial pathname that contains no colons. See also: **leafname, partial pathname**.

pattern: A literal text pattern (such as /ABCDEFG/), or a regular expression. Patterns are a case of selection and always appear between the pattern delimiters /.../ or \...\.

pipe: The command terminator | is the pipe (or pipeline) symbol. It causes the output of the preceding command to be used as the input for the subsequent command. (See Chapter 5, Table 5-1.)

position: In editing commands, position refers to the location of the **insertion point**.

prefix: The directory portion of a filename.

prerequisite file: In Make, the files that must exist or be up-to-date before the target file can be built.

project: In Projector, a set of files that may include other projects (subproject).

project directory: The directory in which Projector maintains all the project-management information about a given project.

project file: In Projector, the file (always named ProjectorDB) in which an entire project is maintained. There is one and only one project file within every project directory.

project information: Information maintained by Projector on a per-project basis, including: author, last modification date, and comment.

project name: In Projector, the name of a project as well as the name of the directory containing that project.

ProjectorDB: In Projector, the name of the database file in which Projector stores all information about projects, their revision trees, revisions, and branches.

project tree: In Projector, the set of mounted projects

pseudo-filename: Any device name that you can use in place of a filename but that has no disk file associated with it. Any command can open a pseudo-filename. These are most often used for I/O redirection.

quotes: A set of characters that literalize the enclosed characters, used for disabling special characters. The quote symbols are '...', "...", \...\, and /.../. The escape character, ∂, quotes the character that follows it.

reference: The location within one module that contains the address of another module or entry.

regular expressions: A language for specifying text patterns, using a special set of metacharacters. (See Appendix B, Table B-2.)

regular expression operators: A special set of metacharacters used in regular expressions and filename generation. (See "Pattern Matching" in Chapter 6.)

resource: Data or code stored in a resource file and managed by the Macintosh Resource Manager.

resource attribute: One of several characteristics, specified by bits in a resource reference, that determine how the resource should be dealt with.

resource compiler: A program that creates resources from a textual description. The MPW resource compiler is named Rez.

resource description file: A text file that can be read by the resource compiler and compiled into a resource file. The resource decompiler disassembles a resource file, producing a resource description file as output.

resource file: Common usage for the resource fork of a Macintosh file.

resource fork: The part of a file that contains data used by an application, such as menus, fonts, and icons. An executable file's code is also stored in the resource fork.

revision: In Projector, an instance of a file in a project. A new revision is created each time a modified file is checked in.

revision information: Information maintained by Projector on a per-revision basis. Also known as the current state of a revision. For unlocked revisions this includes: Author, Creation date, Comment, and Task. For locked revisions the information is: Author (person who checked out the file), Check-out Date, and Task.

revision tree: In Projector, the composite history of a file; that is, all the revisions and branches made to a file. The revision tree for a file can be displayed via the Status command or by double clicking a filename in the Project hierarchy frame of the Check In and Check Out windows.

root: In a makefile, a top-level target that is not a prerequisite of any other target.

scaling: In graphics, to shrink or expand an image. See Appendix G: The Graf3D Library for more information.

scope: In Projector, the current project.

script: An ordinary text file (type TEXT) containing a series of commands. The entire file can be executed by entering the filename. A script is also referred to as a *command file* or *command script.*

segment: One of several parts into which the code of an application may be divided. Not all segments need to be in memory at the same time.

selection: A series of characters, or a character position, at which the next editing operation will occur. Selected characters are inversely highlighted in the active window, and outlined in other windows. A selection is used as an argument to most editing commands and can be specified by using a special set of selection operators. (See Appendix B, Table B-1.)

signal: An event that diverts program control from its normal sequence. (See Chapter 12.)

signature: Each Macintosh application has its own unique signature (or creator). For example, creating a file with the type DFIL and signature DMOV tells the Font/DA Mover that this file contains desk accessories. See the Finder Interface chapter of *Inside Macintosh.*

simple command: Any command consisting of a single keyword followed by zero or more parameters.

standard error: See "diagnostic output."

standard input: Input to a command, usually typed directly into the active window (the console).

standard output: Output produced by most commands that is returned to an open file, usually the window in which its command or program was typed.

Startup file: A special command file containing commands that are executed each time the Shell is launched. Startup executes a second command file called UserStartup.

status panel: The panel in the lower left corner of the Worksheet window. The status panel shows what command MPW is executing. Clicking in the status panel is equivalent to pressing the Enter key.

status value: A code returned by commands in the Shell variable {Status}. Zero indicates successful completion of the previous command, and other values usually indicate an error.

structured command: Any command that controls the order in which other commands are executed. For and If are examples of structured commands. All structured commands are built into MPW and usually have more than one keyword. See also **simple command.**

subproject: In Projector, any project contained within another project. Subprojects may, in turn, contain other subprojects.

target selection: The current selection in the target window, represented by the § character.

target file: In Make, a file that is to be rebuilt and that depends on one or more prerequisite files.

target window: The second window from the front—this is the default target for editing commands that are entered in the active window. The Shell variable {Target} always contains the name of the current target window.

task: In Projector, a short description of the task that a user accomplished with a revision.

{task}: In Projector, the name of the current task. It appears in the Check Out and Check In windows as the default task.

tool: See **MPW Tool.**

type declaration: In the resource compiler, a statement that specifies the pattern for any associated resource data by indicating data types, alignment, size, and placement of strings.

translation: In graphics, movement anywhere in three-dimensional space. See Appendix G: The Graf3D Library for more information.

trunk: In Projector, the main sequence of revisions to a file. Branches from any revision are always named and numbered with respect to the trunk.

users: In Projector, those persons with access to the files of a project.

{user}: In Projector, the name of the current user. Projector logs this name with all transactions. You can override this name by specifying a different name with the -u option available in all Projector commands.

user interface: The system or set of conventions by which the user interacts with software. In addition to the standard Macintosh mouse-and-menu interface, MPW includes both a command language and a dialog user interface (Commando).

word: A single, blank-separated element in a command. A command name and each of its parameters are separate words in the command language.

Worksheet window: The main work area in MPW; the window usually used as the console.

Index

difference from MPW tools 351
running outside Shell 129
argc 357
argv 357
arrangement of MPW file 40
Assembler 33
attributes
  in Rez 310

B

backquote key 144
backquotes 144
backslashes 148, 149
blank interpretation 150
'BNDL' resource 243
branching
  definition 199
  in Projector 218
bucket counts
  performance measurement 452
buffering
  buffer initialization procedure 365
  MPW Shell input/output 358
  standard I/O buffering 366
  stdio 364
  warning about use of file descriptors with
    FILE variable 366
Build menu
  introduction 49
  tutorial 50
  modifying 259
building
  desk accessories
    limitations 258
    location of code 258
    result code 257
  desk accessory 254
    steps in building 251
  driver 254
    steps in building 251
  drivers
    calling sequence 257
    structure of 257
  MPW tools 350
  stand-alone code resources 248

steps in creating 248
building a program
  introduction 49
  new program 54
  steps 241
bulldozer 194
bundle bit 243

C

C compiler 34
C++ translator 34
case sensitivity 186
Caution
  on use of global variables 254
CC 170
changing directories 101
characters 145
  asterisk
    in regular expressions 187
  case sensitivity of 186
  character list 186
  colon 183
  current selection 162, 167
  escape 148
  exclamation mark 179
    in filename generation 145
  infinity 167
  integral character 227
  invisible 189
  literalizing 185
  negation 186
  newline 178
  plus sign
    in regular expression 187
  ® oeprator 188
  regular expression operators
    table of 184
  returns in command definitions 155
  slashes 182
  special 147–48
    use of 160
close function 370
code resources
  controlling the numbering of 290
  'CODE' resources 244

APPLE COMPUTER, INC. SOFTWARE LICENSE

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Apple Software") and related documentation are licensed to you by Apple. You own the disk on which the Apple Software is recorded but Apple and/or Apple's Licensor(s) retain title to the Apple Software and related documentation. This License allows you to use the Apple Software on a single Apple computer and make one copy of the Apple Software in machine-readable form for backup purposes only. You must reproduce on such copy the Apple copyright notice and any other proprietary legends that were on the original copy of the Apple Software. You may also transfer all your license rights in the Apple Software, the backup copy of the Apple Software, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Apple Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Apple Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Apple Software in whole or in part. You may not electronically transmit the Apple Software from one computer to another or over a network.

3. Support. You acknowledge and agree that Apple may not offer any technical support in the use of the Software.

4. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Apple Software and related documentation and all copies thereof. This License will terminate immediately without notice from Apple if you fail to comply with any provision of this License. Upon termination you must destroy the Apple Software and related documentation and all copies thereof.

5. Export Law Assurances. You agree and certify that neither the Apple Software nor any other technical data received from Apple, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States.

6. Government End Users. If you are acquiring the Apple Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:
    (i) if the Apple Software is supplied to the Department of Defense (DoD), the Apple Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Apple Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and
    (ii) if the Apple Software is supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Apple Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

7. Limited Warranty on Media. Apple warrants the disks on which the Apple Software is recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Apple's entire liability and your exclusive remedy will be replacement of the disk not meeting Apple's limited warranty and which is returned to Apple or an Apple authorized representative with a copy of the receipt. Apple will have no responsibility to replace a disk damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKS, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. Disclaimer of Warranty on Apple Software. You expressly acknowledge and agree that use of the Apple Software is at your sole risk. The Apple Software and related documentation are provided "AS IS" and without warranty of any kind and Apple and Apple's Licensor(s) (for the purposes of provisions 8 and 9, Apple and Apple's Licensor(s) shall be collectively referred to as "Apple") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. APPLE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE APPLE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE APPLE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE APPLE SOFTWARE WILL BE CORRECTED. FURTHERMORE, APPLE DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE APPLE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE APPLE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

9. Limitation of Liability. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL APPLE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE APPLE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.
In no event shall Apple's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Apple Software.

10. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

11. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Apple Software and related documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Apple.

7/15/91
001-0158-A