Sistemi di sviluppo
Documentazione tecnica
Manualistica ed editoria specializzata

ULISSE s.r.l.
Servizio Distribuzione Apple
Via Pacini, 22
20131 Milano
Tel. (02) 2367783-2367406

su licenza esclusiva
Apple Computer S.p.A.

Macintosh Programmer's Workshop Reference

Version 1.0

MPW00

Macintosh Programmer's Workshop Reference

October 3, 1986

Apple Technical Publications

This document contains preliminary information. It does not include

- *final editorial corrections*
- *final art work*
- *an index.*

It may not include final technical changes.

Contents

Figures and Tables x

Preface ~~xx~~ xvii

Power tools for Macintosh programmers x
What you'll need x
    Hardware requirements x
    System Folder requirements x
    Pascal and C requirements x
    Documentation x
About this ranual x
    Syntax notation x

System Overview ~~xx~~ 1

A road map x
The MPW Shell x
    File handling commands x
    Editing commands x
    Structured commands x
    Other built-in commands x
MPW tools x
    Assembler x
    Pascal tools x
    C Compiler x
    Linker x
    Make x
    Resource Compiler and Decompiler x
    Conversion tools x
Applications x
    ResEdit x
Debugger x
Special command files x
Sample program files x
System Folder x

Overview of MPW files and directories x

Chapter 1 **Getting Started** ~~xx~~ 13

- Installing the system x
  - HD-20 installation x
  - Macintosh XL installation x
  - Installing the system on 800K disks x
- Starting up x
- Editing x
- Giving commands x
  - The Enter key x
  - Executing several commands at once x
  - Terminating a command x
  - The Help command x
- File Handling commands x
- File and window names x
  - Selection specifications x
  - Directories and pathnames x
  - Command search path x
  - Changing directories x
    - An aside: the Alias command x
  - Pathname variables x
  - Wildcards (filename generation) x
- Building a program x
  - Assembling and compiling x
  - Linking x
  - Automating the build process with Make x
- Running an application x

Chapter 2 **Basic Editing** ~~xx~~ 33

- Features x
- File format x
- Menu commands x
  - Apple menu x
  - File menu x
  - Edit menu x
  - Find menu x
  - Format menu x
  - Windows menu x
  - User-defined menus x
- Editing with the command language x

Chapter 3 **Using the Command Language** ~~xx~~ 45

Overview x
Types of commands x
Entering and executing commands x
Structure of a command x
- Command name x
- Parameters x
- Command terminators x
- Comments x
- Simple versus structured commands x

Running an application outside the Shell environment x
Command files x
Special command files x
- The Startup and UserStartup file x
- Suspend,Resume, and Quit x

Command aliases x
- Executable error messages x

Variables x
- Predefined variables x
- Variables defined in the Startup file x
- Parameters to command files x
- Defining and redefining variables x
- Exporting variables x

Command substitution x
Quoting special characters x
How commands are interpreted x
Structured commands x
- Control loops x
- Processing command parameters x
- Expressions x

Filename generation x
Redirecting input and output x
- Standard input x
  - Terminating input with Command-enter x
- Standard output x
- Diagnostic output x
- Pseudo-filenames x

Defining your own menu commands x
Sample command files x
- "Add Menu as Group" x

Chapter 4 **Advanced Editing** ~~xx~~ 21

Editing Commands x
Selections xx

Current selection x
Selection by line number x
Position x
Extending a selection x
Pattern x
Pattern Matching (Using Regular Expressions) x
Character expressions x
"Wildcard" operators x
Repeated Instances of regular expressions x
Tagging regular expressions with the ® operator x
Matching a pattern at the beginning or end of a line x
Inserting invisible characters x
Note on forward and backward searches x
Some Useful Examples x
Transfering DumpObj output x
Finding a whole word x

Chapter 5 Editing Resources With ResEdit ~~xx~~ 96

About ResEdit x
Uses x
Extensibility x
Using ResEdit x
Working with files x
Working within a file x
Working within a resource type x
Editing individual resources x
'CURS' resources x
'DITL' resources x
'FONT' resources x
'ICN#' resources x
Creating a resource template x

Chapter 6 Resource Compiler and Decompiler ~~xx~~ 114

About the Resource Compiler and Decompiler x
Resource Decompiler x
Standard type declaration files x
Using Rez and DeRez x
Structure of a resource description file x
Sample resource description file x
Resource description statement x
Syntax notation x
Special terms x
Include x
Syntax x

        Resource Attributes x
    Read x
        Syntax x
        Description x
    Data x
        Syntax x
        Description x
    Type x
        Syntax x
        Description x
        Data-type specifications x
            Numeric types x
            Boolean type x
            Character type x
            String types x
            Point and rectangles x
            Fill and align types x
            Fill specifications x
            Align specifications x
            Array type x
            Switch type x
    Resource x
        Syntax x
        Description x
        Data statements x
            Switch data x
            Array data x
        Sample resource definition x
            Symbolic names x
    Preprocessor directives x
        Variable definitions x
        Include directives x
        If-Then-Else processing x
    Resource description syntax x
        Numbers and literals x
        Expressions x
        Variables x
        Strings x
            Escape characters x

Chapter 7 Building a Program 144

    Overview of the build process x
    Structure of a Macintosh application x
    Linking x
        What to link with x

Linking together code written in different languages x
File types and creators x
Putting together an MPW tool x
Putting together a desk accessory or driver x
Linking a desk accessory or driver x
The desk accessory resource file x
Using Make x
Format of a makeFile x
Dependency rules x
Double-*f* dependency rules x
Default rules x
Built-in default rules x
Directory dependency rules x
Variables in makefiles x
Shell variables x
Defining variables within a makefile x
Built-in Make variables x
Comments x
Quoting x
Executing Make's output x
Debugging makefiles x
An example x
More about linking x
Linker functions x
Segmentation x
Setting resource attributes x
Controlling the numbering of code resources x
Resolving symbol definitions x
Multiple external symbol definitions x
Unresolved external symbols x
Linker location map x
Optimizing your links x
Library construction x
Using Lib to build a specialized library x
Removing unreferenced modules x
Guidelines for choosing files for a specialized library x

Chapter 8 Debugging ~~xx~~ 181

About MacsBug x
Installing MacsBug x
Theory of operation (a technical aside) x
Using MacsBug x
The MacsBug command language x
Numbers x

Strings x
Symbols x
Expressions x
Commands x
General commands x
Memory commands x
Break commands x
A-Trap commands x
Heap Zone commands x
Disassembler commands x
Summary x

Chapter 9 Command Reference ~~xx~~ 206

Command prototype x
AddMenu — Add menu item x
Adjust — Adjust lines x
Alert — Display alert box x
Alias — Define and write command aliases x
Align — Align text to left margin x
Asm — 68xxx Macro Assembler x
Beep — Generate tones x
Begin...End — Group commands x
Break — Break from For or Loop x
C — C Compiler x
Canon — Canonical spelling tool x
Catenate — Concatenate files x
Clear — Clear the selection x
Close — Close a window x
Compare — Compare text files x
Confirm — Display confirmation dialog x
Continue — Continue with next iteration of For or Loop x
Copy — Copy selection to Clipboard x
Count — Count lines and characters x
Cut — Copy selection to Clipboard and delete it x
CvtObj — Convert Lisa object files to MPW object files x
Date — Write the date and time x
Delete — Delete files and directories x
DeleteMenu — Delete user-defined menus and items x
DeRez — Resource Decompiler x
Directory — Set and write the default directory x
DumpCode — Write formatted code resources x
DumpObj — Write formatted object file x
Duplicate — Duplicate files and directories x
Echo — Echo parameters x
Eject — Eject volumes x

Entab — Convert runs of spaces to tabs x
Equal — Compare files and directories x
Erase — Initialize volumes x
Evaluate — Evaluate an expression x
Execute — Execute a command file in the current scope x
Exit — Exit from command file x
Export — Make variables available to programs x
FileDiv — Divide a file into several smaller files x
Files — List files and directories x
Find — Find and select a text pattern x
Font — Set font characteristics x
For... — Repeat commands once per parameter x
Help — Display summary information x
If... — Conditional command execution x
Lib — Combine object files into a library file x
Link — Link an application, tool, or resource x
Loop...End — Repeat command list until Break x
Make — Build up-to-date version of a program x
MDSCvt — Convert MDS Assembler source x
Mount — Mount volumes x
Move — Move files and directories x
New — Open a new window x
NewFolder — Create a directory x
Open — Open a window x
Parameters — Write parameters x
Pascal — Pascal Compiler x
PasMat — Pascal program formatter ("pretty-printer") x
PasRef — Pascal cross-referencer x
Paste — Replace selection with contents of the Clipboard x
Print — Print text files x
Rename — Rename files and directories x
Replace — Replace the selection x
Request — Request text from a dialog x
Rez — Resource Compiler x
RezDet — The resource detective x
Save — Save windows x
Search — Search files for a pattern x
Set — Define and write Shell variables x
SetFile — Set file attributes x
Shift — Renumber command file positional parameters x
Tab — Set a window's tab value x
Target — Make a window the target window x
TLACvt — Convert Lisa TLA Assembler source x
Unalias — Remove aliases x
Unmount — Unmount volumes x

Unset — Remove Shell variables x
Volumes — List mounted volumes x
Windows — List windows x

Appendix A **Macintosh Workshop Files** ~~xx~~ 385
Distribution files (annotated) x
HD-20 configuration x
800K disk configuration x

Appendix B **Summary of Selections and Regular Expressions** ~~xx~~ 402
Selections x
Regular expressions x
Option key characters x

Appendix C **MPW Characters** ~~xx~~ 406

Appendix D **Resource Description Syntax** ~~xx~~ 410

Appendix E **File Types, Creators, and Suffixes** ~~xx~~ 419

Appendix F **Writing an MPW Tool** ~~xx~~ 421
Shell facilities x
Parameters x
Shell variables x
Standard I/O channels x
Status results x
Signals x
Exit processing x
Restrictions x
Initialization x
Memory management x
Heap x
Stack x
Windows, graphics, and events x
Style x
User interface x

Appendix G	Writing a Desk Accessory or Other Driver Resource ~~xx~~ 433
Appendix H	Object File Format ~~xx~~ 438
Appendix I	In Case of Emergency ~~xx~~ 449
Glossary	
Index	

Figures and tables

System Overview xx

Figure 1	Steps in program development xx
Figure 2	Setup of MPW folders and files xx

Chapter 1 Getting Started xx

Figure 1-1	Worksheet windows xx
Figure 1-2	Press enter to execute selected text xx
Figure 1-3	Help summaries xx
Figure 1-4	Hierarchical directory structure xx
Figure 1-5	Setting the default directory xx
Figure 1-6	Executing Make output for samples xx
Table 1-1	Basic file handling commands xx

Chapter 2 Basic Editing xx

Figure 2-1	Text selected with Find command xx
Figure 2-2	Text highlighted in the active window and target window xx

Chapter 3 Using the Command Language xx

Figure 3-1	Trafficking in variables xx
Figure 3-2	Standard input and output xx
Figure 3-3	Redirecting diagnostic output xx
Table 3-1	Command terminators xx
Table 3-2	Variables defined by the Shell xx
Table 3-3	Variables defined in the Startup file xx
Table 3-4	Parameters to command files xx
Table 3-5	Special characters and words xx
Table 3-6	Quotes xx
Table 3-7	Structured commands xx
Table 3-8	Expression operators xx
Table 3-9	Filename generation operators xx
Table 3-10	I/O redirection xx
Table 3-11	Pseudo-filenames xx

Chapter 4 Advanced Editing xx

Figure 4-1	A selection specification xx
Figure 4-2	Selections in two windows xx

Table 4-1 Editing commands xx
Table 4-2 Selection commands xx
Table 4-3 Regular expression operators xx

Chapter 5 **Editing Resources With ResEdit xx**
Figure 5-1 A disk volume window xx
Figure 5-2 A file window xx
Figure 5-3 A resource type window xx
Figure 5-4 Editing 'CURS' resources xx
Figure 5-5 Font editor window xx
Figure 5-6 'ICN#' window xx
Figure 5-7 Window template data xx

Chapter 6 **Resource Compiler and Decompiler xx**
Figure 6-1 Rez and DeRez xx
Figure 6-2 Creating a resource file xx
Figure 6-3 Padding of literals xx
Figure 6-4 Internal representation of a Pascal string xx
Table 6-1 Resource description file expansion operators xx
Table 6-2 Resource Compiler escape sequences xx

Chapter 7 **Putting Together an Application, MPW Tool, or Desk Accessory**
Figure 7-1 Building a program xx
Figure 7-2 Linking xx
Figure 7-3 Building a desk accessory with DRVRRuntime xx
Table 7-1 Files to link with xx
Table 7-2 File types and creators xx
Table 7-3 Makefile summary xx

Appendix B **Summary of Selections and Regular Expressions xx**
Table B-1 Selections xx
Table B-2 Regular expressions xx

Appendix C **MPW Characters xx**
Table C-1 MPW operators xx

Appendix E **File Types, Creators, and Suffixes xx**
Table E-1 File types and creators xx

Appendix F Writing an MPW Tool xx

Figure F-1	Parameters in C and Pascal xx
Figure F-2	I/O buffering xx
Figure F-3	Memory map xx

Preface

Power Tools for Macintosh Programmers

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

- MPW Shell (the programming environment)
- 68xxx Assembler
- Linker
- Resource Editor
- Resource Compiler and Decompiler
- Debugger

The system also includes many other tools for creating and manipulating text and resource files. The following Macintosh Workshop products are separately available:

- **Macintosh Programmer's Workshop Pascal** provides the additional tools, interfaces, and libraries you need to develop applications, tools, and desk accessories in Pascal.
- **Macintosh Programmer's Workshop C** provides the Green Hills Software C Compiler, along with the interfaces and libraries needed to develop applications, tools, and desk accessories in C.

- **MacApp** is an expandable "generic application." MacApp 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.

The entire MPW system is outlined in detail in the "System Overview" section that follows.

The Macintosh Programmer's Workshop provides numerous advantages over previous development systems. Among these advantages are

- Integration—the various components of the MPW system all run within the MPW Shell environment. The integrated environment enables you, for example, to automate build procedures for programs.
- Command scripting—in addition to menu commands MPW provides a full command language. You can combine any series of commands into a command file (or "script") 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 create your own integrated tools to run within the Shell environment. You can also add your own menu commands to the Shell; these commands can be command files, integrated tools, or stand-alone applications.

Taken together, these features add up to a level of integration, power, and ease of use not found in any previous microcomputer-based development system.

What you'll need

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

Hardware requirements

The Macintosh Workshop runs on the Macintosh Plus, the Macintosh 512K and 512K Enhanced, and the Macintosh XL. The system runs on both the original 64K ROMs and the newer 128K ROMs. Apple's Macintosh peripherals, including the LaserWriter™, are supported.

The Workshop requires a minimum disk storage of 1.6 Mbytes (two 800K disks). Use of a hard disk is recommended but not required—with the minimum disk configuration, you can use only one language at a time, without swapping disks. A Macintosh Plus with an Apple Hard Disk 20™ is the recommended configuration.

Both 400K (nonhierarchical) and 800K (hierarchical file system) disks are supported. The software is shipped on 400K disks. Hard disks may be either hierarchical (HFS) or non-HFS volumes, but using non-HFS volumes is more awkward.

System Folder requirements

The System Folder provided with MPW includes version 3.2 of the System file and version 5.3 of the Finder. *System file 3.2 is required for MPW.* In addition, the following versions of the printer drivers are required:

- Laser Prep 3.1
- ImageWriter 2.3
- AppleTalk® ImageWriter™ 2.3

These files are available on version 1.1 or later of the *System Tools* disk, and on version 1.0 or later of the *Printer Installation* disk.

◆ *HD-20 Startup Disk Users:* If you are using a 512K Macintosh with the 64K ROM and an HD-20 (HFS) startup disk, you must use version 1.1 or a later version of the *HD-20 Startup* disk.

Pascal and C requirements

MPW Pascal requires most of the available memory in 512K systems.

MPW C requires a Macintosh Plus or a 1-Mbyte Macintosh XL.

Documentation

All programmers will need Volumes I–III of *Inside Macintosh* (published by Addison-Wesley, 1985), the definitive guide to the Macintosh operating system and user-interface toolbox. In order to program for the Macintosh Plus, you'll also need Volume IV of *Inside Macintosh*. If you need to understand and control the numeric environment, you'll need the *Apple Numerics Manual*, a guide to the Standard Apple Numeric Environment (SANE). Lastly, you'll need the appropriate documentation for the programming language you'll be using:

- **Assembly Language:** *Macintosh Programmer's Workshop Assembler Reference*. This manual is included in your Macintosh Programmer's Workshop package. You'll also need the appropriate microprocessor documentation from Motorola.
- **Pascal:** *Macintosh Programmer's Workshop Pascal Reference*. This manual is available as part of a separate MPW product.
- **C:** *Macintosh Programmer's Workshop C Reference*. This manual 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 & D. Ritchie, or a similar C manual.

About this manual

This book describes the MPW development system, including the Shell and tools. This manual 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 manuals listed above.

Syntax notation

The following syntax notation is used to describe Macintosh Workshop commands:

terminal	Plain text indicates a word that must appear in the command exactly as shown. Special symbols (-, §, &, and so on) must also be entered exactly as shown.

nonterminal	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.
repeated...	Ellipses (...) indicate that the preceding item can be repeated one or more times.
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 MPW.Help file. (See "The Help Command" in Chapter 1.)

Filenames and command names are not sensitive to case. By convention, they are shown with initial capital letters.

Terms printed in boldface appear in the glossary.

System Overview

A road map xx

The MPW Shell xx

File handling commands x
Editing commands x
Structured commands x
Other built-in commands x

MPW tools xx

Assembler x
Pascal tools x
C Compiler x
Linker x
Make x
Resource Compiler and Decompiler x
Conversion tools x

Applications xx

ResEdit x

Debugger xx

Special command files xx

Sample program files xx

System Folder xx

Overview of MPW files and firectories xx

This section outlines the program development process and contains a general description of the various parts of the Macintosh Programmer's Workshop.

A Road Map

Figure 1 illustrates the typical stages of program development. Examples in Chapter 1 recapitulate these steps, using the ***Inside Macintosh*** sample program as an example.

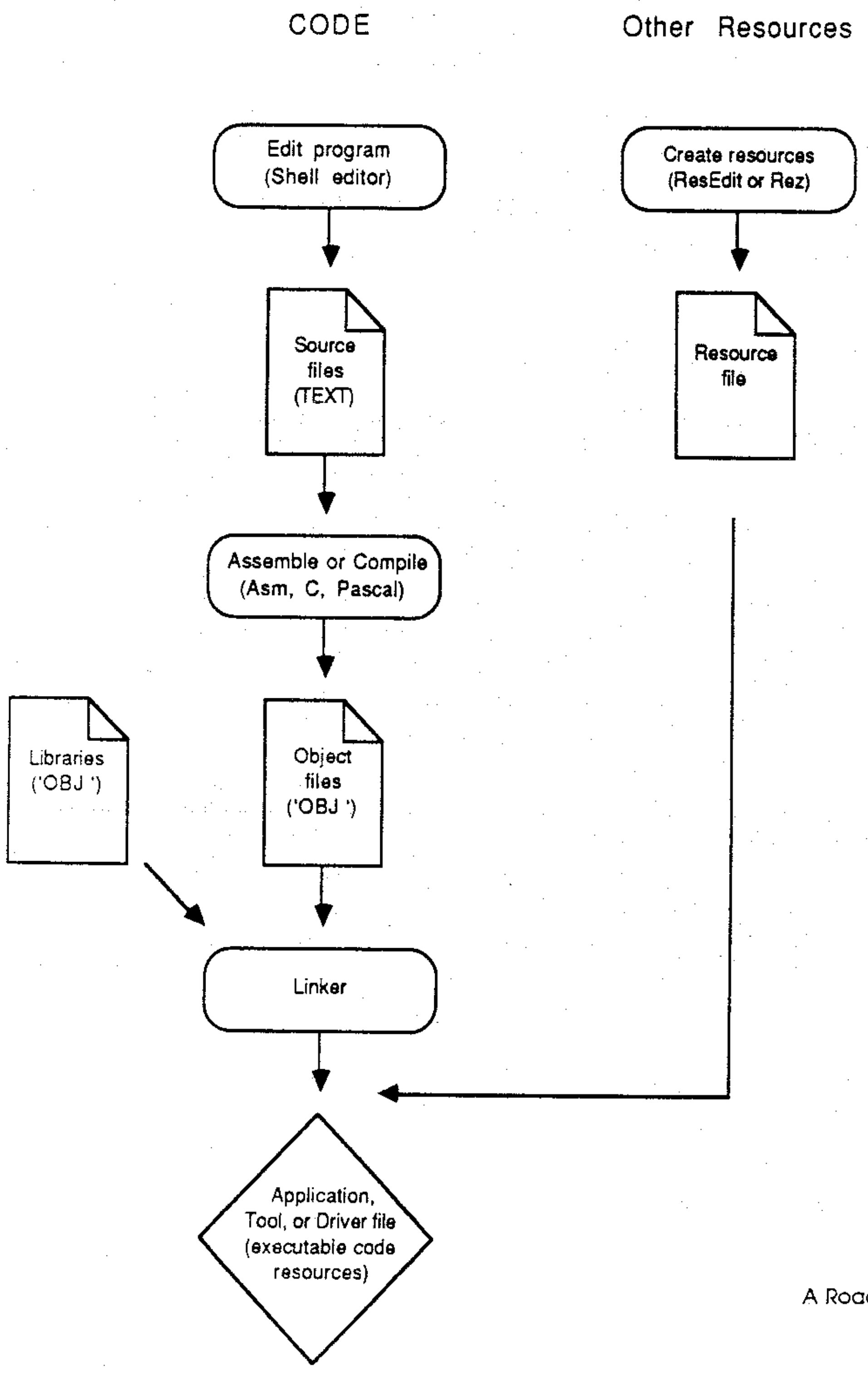

CODE
Other Resources
Edit program (Shell editor)
Create resources (ResEdit or Rez)
Source files (TEXT)
Resource file
Assemble or Compile (Asm, C, Pascal)
Libraries ('OBJ ')
Object files ('OBJ ')
Linker
Application, Tool, or Driver file (executable code resources)

Figure 1
Steps in program development

The rest of this section describes the various parts of the Macintosh Workshop, and how they relate to the development process. (See Chapter 1, "Getting Started," for information about entering MPW commands.)

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 Workshop—the Assembler, Compilers, and other tools described below—all operate within the Shell environment. These tools can perform input and output to files and to Shell windows.

The Shell combines a command language and a text editor. You can enter commands in any window, or execute them by using menus and dialogs. The command language provides text editing and program execution functions, including parameters to programs, command file (scripting) capabilities, input/output redirection, and structured commands.

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 2 and 4.)
- A **command interpreter** that interprets and executes commands that you enter in a window or read from a file. (See Chapters 3 and 9.)
- **Built-in commands**—besides editing commands, these include commands for handling files without returning to the Finder, processing variables, program control flow, and more. (See Chapter 3.)

File handling commands

The MPW Shell provides the following built-in commands for manipulating files and directories without having to exit to the Finder:

Catenate	concatenate files
Close	close a window
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
Files	list files and directories
Mount	mount volumes
Move	move files and directories
New	open a new window
NewFolder	create a directory
Open	open a window
Rename	rename files and directories
Save	save windows
SetFile	set file attributes
Target	make a window the target window
Unmount	unmount volumes
Volumes	list mounted volumes
Windows	list windows

Editing commands

Besides the Macintosh's usual mice-and-menus editing capabilities, a number of built-in editing commands are provided. You can use these commands both interactively and in command files. Editing commands feature the use of **regular expressions**, a set of special operators that forms a powerful language for defining text patterns.

Adjust	adjust lines
Align	align text to left margin
Clear	delete the selection
Copy	copy selection to the Clipboard
Cut	copy selection to the Clipboard and delete the selection
Find	find and select a text pattern
Font	set a window's font characteristics
Paste	replace selection with contents of the Clipboard
Replace	replace the selection
Tab	set a window's tab value

Structured commands

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

If...	conditional command execution
For...	repeat commands once per parameter
Loop...End	repeat commands until Break
Begin...End	group commands
Break	break from For or Loop
Continue	continue with next iteration of For or Loop
Exit	exit from command file

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
Date	write the date and time
DeleteMenu	delete user-defined menus and items
Echo	echo parameters
Evaluate	evaluate an expression
Execute	execute a command file without affecting variable scope
Export	make variables available to programs
Help	display summary information
Parameters	write parameters
Request	request text from a dialog
Set	define and write Shell variables
Shift	renumber command-file positional parameters
Unalias	remove aliases
Unset	remove Shell variables

MPW tools

MPW tools are programs that run within the Shell environment. The following tools are provided with the Macintosh Workshop; several are described in more detail in the following sections.

Asm	68xxx Macro Assembler
C	C Compiler (available as a separate product)
Canon	canonical spelling tool
Compare	compare text files
Count	count lines and characters
CvtObj	convert Lisa® object files to MPW object files
DeRez	Resource Decompiler
DumpCode	dump code resources
DumpObj	dump object files
FileDiv	divide a file into several smaller files
Lib	combine object files into a library file
Link	link an application, tool, or resource
Make	program maintenance tool
MDSCvt	convert MDS Assembler source
Pascal	Pascal Compiler (available as a separate product)
PasMat	Pascal program formatter (part of the MPW Pascal product)
PasRef	Pascal cross-referencer (part of the MPW Pascal product)
Print	print text files
Rez	Resource Compiler
RezDet	the resource detective
Search	search files for a pattern
TLACvt	convert Lisa TLA Assembler source

Assembler

The Assembler translates 68000, 68010, and 68020 assembly-language programs into object code. 68881 floating-point instructions and 68851 memory-management instructions are also supported. The Assembler provides powerful macro facilities, code and data modules and entry points, local labels, and (optional) optimized instruction selection. Assembly-language interfaces are provided to the "Inside Macintosh" libraries (including the 128K ROM). Other libraries and example files are also provided.

Pascal tools

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

- Pascal Compiler
- Pascal cross-referencer (PasRef)
- Pascal program formatter (PasMat)
- Pascal runtime library

- Pascal interfaces to the "Inside Macintosh" routines (including the 128K ROM routines)
- sample programs

Macintosh Workshop Pascal is an improved version of Lisa Pascal. The improvements include SANE numerics, access to C functions and global data, arbitrary-length identifiers, and Object Pascal extensions.

C Compiler

The C Compiler is also provided as part of a separate product, *MPW C*, which includes the following:

- C Compiler
- Standard C Library
- C interfaces to the "Inside Macintosh" libraries (including the 128K ROM routines)
- sample programs

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. Most Standard C Library functions, including character and string processing, memory allocation, and formatted input/output are also provided.

Linker

The Linker combines object code files and resources into executable programs, including only the code and data modules that are referenced. The Linker replaces the code segments in an existing resource file, without disturbing other resources in the file. An option directs the Linker to produce a link map as a text file. A separate tool, Lib, provides library manipulation.

Make

The Make tool simplifies software contruction 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.

Resource Compiler and Decompiler

The Resource Compiler (Rez) reads a textual description of a resource and converts it into a resource file. The Resource Decompiler (DeRez) converts resources into a textual representation that can be edited in the Shell, and recompiled with Rez—DeRez can be used to create Resource Compiler input from any existing resource files. Rez and DeRez use 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.

Conversion tools

TLACvt converts Lisa Workshop Assembler (TLA) source files to MPW Assembler source files. CvtObj converts Lisa Workshop object files to the MPW object file format.

MDSCvt converts Macintosh 68000 Development System (MDS) Assembler source files to MPW Assembler source files.

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

Applications

Applications are stand-alone programs that can be launched from the Shell, but that execute outside the Shell environment. A single application, ResEdit, is 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, such as MacPaint or MacWrite, can be executed from the MPW Shell.

ResEdit

ResEdit is an interactive, graphically based resource editor for creating, editing, copying, and pasting resources. MPW Pascal includes a set of extended Resource Manager routines that make it possible to write your own add-on resource editors for ResEdit.

Debugger

The MacsBug 68000 debugger is provided with the Macintosh Workshop. 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. Another version of MacsBug, in the file MacsBug.XL, is provided for use on the Macintosh XL.

Special command files

Several special command files are provided. These text files contain commands that are read by the MPW Shell at startup and shutdown.

- The **Startup file** is a command file containing a startup script that is run each time you start the MPW Shell. It, in turn, executes a file called **UserStartup**, which you can use to customize the Workshop. The Startup file is discussed in detail in Chapter 3.
- The **Suspend** and **Resume** files are command files that preserve the state of the Shell environment while a stand-alone application is executing. The **Quit** file allows you to save the state of the Shell environment when you exit to the Finder.

Sample program files

Source files are provided for the sample application from *Inside Macintosh*, as well as for a sample MPW tool and a sample desk accessory. Assembly-language versions of these programs are contained in the folder AExamples. MPW Pascal and MPW C also include Pascal and C versions of the sample files, in the folders PExamples and CExamples. The Examples folders also contain instruction files and makefiles for building the sample programs; these are discussed in the next chapter.

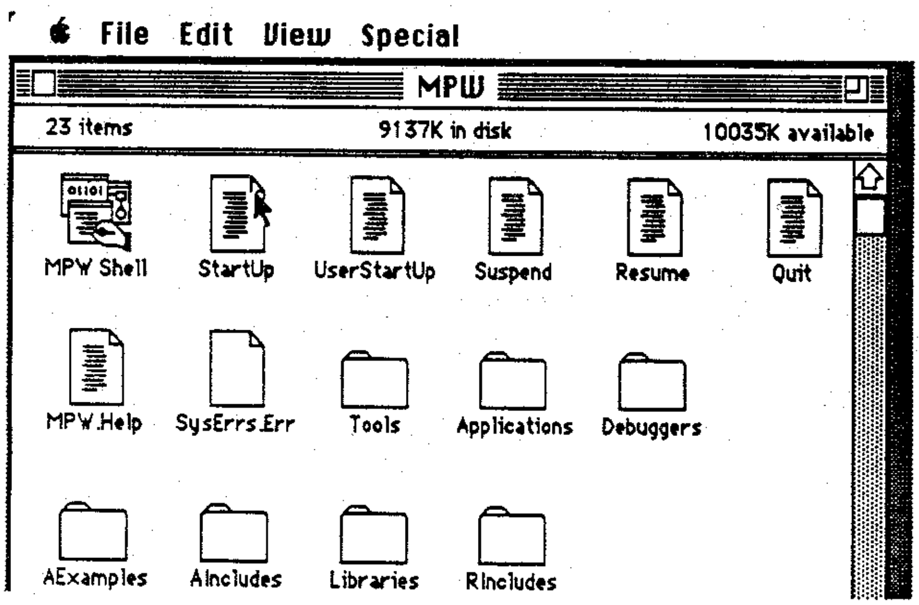
System Folder

The latest versions of the System file and Finder have been provided for use with MPW and on your application disks. Several fonts have been removed from the System file, to reduce its size for use with systems without a hard disk.

❖ *Note:* System file version 3.2 is required for use of MPW. MPW also requires the printer drivers provided on the *System Tools 1.1* or *Printer Installation 1.1* disk.

Overview of MPW files and directories

Appendix A contains a complete, annotated list of all of the Macintosh Workshop files. It also describes the recommended setup of files on an HD-20 or set of 800K disks. Figure 2 shows the initial setup of your MPW folders and files on an HD-20. (The Pascal and C systems are included.)



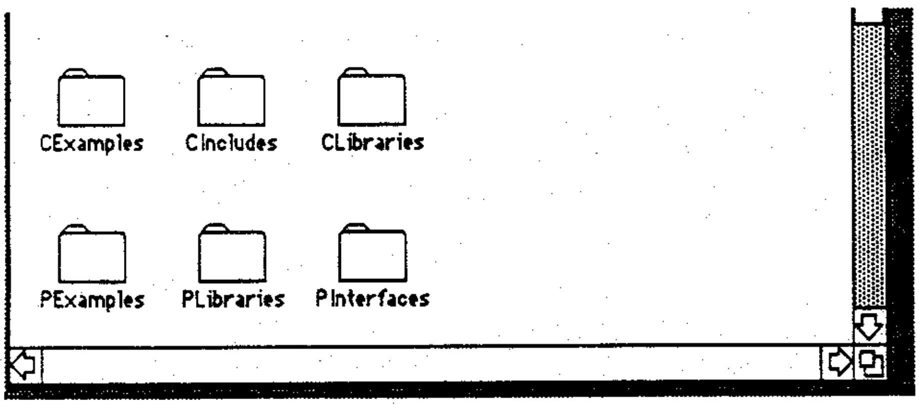


Figure 2
Setup of MPW folders and files

For important information about setting up your MPW system, see "Installing the System" in Chapter 1.

Chapter 1

Getting Started

Installing the system xx

HD-20 installation x
Macintosh XL installation x
Installing the system on 800K disks x

Starting up xx

Editing xx

Giving commands xx

The Enter key x
Executing several commands at once x
Terminating a command x
The Help command x

File handling commands xx

File and window names xx

Selection specifications x
Directories and pathnames x
Command search path x
Changing directories x
    An aside: The alias command x
Pathname variables x
Wildcards (filename generation) x

Building a program xx

Assembling and compiling x
Compiling Resources x

Linking x
Automating the build process with make x
Running an application xx

This chapter introduces the use of the Macintosh Programmer's Workshop, and briefly describes the steps in developing a Macintosh application. It shows you how to assemble, link, and run a simple application by using examples contained in the "Examples" folders. Example files included for assembly language, Pascal, and C.

Installing the system

The Macintosh Programmer's Workshop is shipped on five 400K disks; the Pascal and C systems occupy another two disks each. This section describes how to install the Macintosh Workshop files onto the following disks:

- the Apple HD-20, which uses the hierarchical file system (HFS)
- a Macintosh XL hard disk, which uses the "flat" (nonhierarchical) file system
- a set of 800K floppy disks

Appendix A, "Macintosh Workshop Files," shows the recommended layout of files on an HD-20 or 800K disk system. HFS pathname rules are explained in this chapter.

❖ *Note:* A command file named Startup is executed by the MPW Shell during initialization. This file defines several Shell variables, including the variables that indicate the location of MPW tools, applications, include files, and libraries. The file originally named "Startup" works with the standard configuration (HD-20, hierarchical file system). Special Startup files have been provided for use with nonhierarchical file systems and 800K-only systems—see the instructions that follow.

Note also that System file version 3.2 is required for using MPW. (A version of this file, with several fonts removed, is shipped with the system.)

HD-20 installation

Use the Finder to do the following:

1. Create a folder named MPW on the hard disk.
2. Copy the contents of all five disks (*except* the System Folder on the MPW1 disk) to folder MPW. If you have Pascal or C, also copy those disks into the MPW folder.
3. Move the entire contents of the folder More Tools to the Tools folder; throw away More Tools.

4. Move the file Asm to the Tools folder. If you have Pascal or C, move the Pascal and C Compilers (named Pascal and C) into the Tools folder. MPW Pascal includes some additional Pascal tools, PasMat and PasRef, which should also be moved into the Tools folder.

❖ *Note:* The files Startup, UserStartup, Suspend, Resume, Quit, MPW.Help, and SysErrs.Err need to be in the same folder as the MPW Shell application, or in the System Folder.

By default, the Macintosh Workshop assumes installation in the MPW folder as described above. Other configurations are possible—but you'll need to make some simple changes to the pathnames defined in the Startup file so that the Shell and tools can find various files.

Macintosh XL installation

A separate Startup file, Startup.XL, has been provided for use with nonhierarchical file systems. To install MPW onto a Macintosh XL, use the Finder to do the following:

1. Rename the file Startup (for example, to Startup.HD20), and rename the file Startup.XL to Startup.
2. Simply copy all of your Macintosh Workshop files onto the hard disk—the distribution files have unique names, so you needn't worry about any name conflicts.

❖ *Note:* The file Sample.r appears with the assembly-language, Pascal, and C examples. All three Sample.r files are identical, so the name conflict can be safely ignored. The file Memory.r also appears with both the Pascal and C examples.

Folders aren't recognized by the nonhierarchical file system, so the arrangement of files in folders is irrelevant to the functioning of the system.

❖ *Note:* The examples in this manual assume a hard disk running HFS. They'll work properly on a Macintosh XL if you leave off any pathname prefixes.

Installing the system on 800K disks

Another Startup file, Startup.800K, has been provided for your use if you are running MPW from 800K disks. To install the system onto a set of 800K disks, use the Finder to do the following:

1. Rename Startup (for example, to Startup.HD20), and rename the file Startup.800K to Startup.
2. Copy the distribution files and folders onto your 800K disks to create the arrangement given in Appendix A under "800K Disk Configuration."

Several configurations of files are possible. We recommend creating an MPW boot disk and a separate disk for each language system you use. The startup disk should contain the System Folder, MPW Shell, RIncludes and Libraries folders, and a Tools folder containing a number of the most useful tools. This disk should remain in the drive while you run MPW. Each of the language systems will also fit onto a single disk (which means that you can use only one language at a time).

With this arrangement, the MPW disk has about 100K bytes free, allowing you to add printer drivers, fonts, or frequently used tools. The language disks each have 300–400K of free space for your source, object, and application files. (You can also move the Examples folders to free up additional disk space.)

❖ *Note:* The examples in this manual assume you are running MPW from an HD-20. To make the examples work on an 800K-only system, you'll have to change some of the pathnames used in the examples, and make sure the correct disks are on-line. (See Appendix A.)

Starting up

❖ *Note:* A small RAM cache (perhaps 32K) is useful when running MPW. Larger caches may be used on the Macintosh Plus with the Assembler and Pascal, but are not recommended when using MPW C. Use of MPW with the Switcher is not recommended.

From the Finder, select and open the MPW Shell icon. The **Worksheet window** (shown in Figure 1-1) 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.

You can also start the Workshop by double-clicking on any Macintosh Workshop text document or tool.

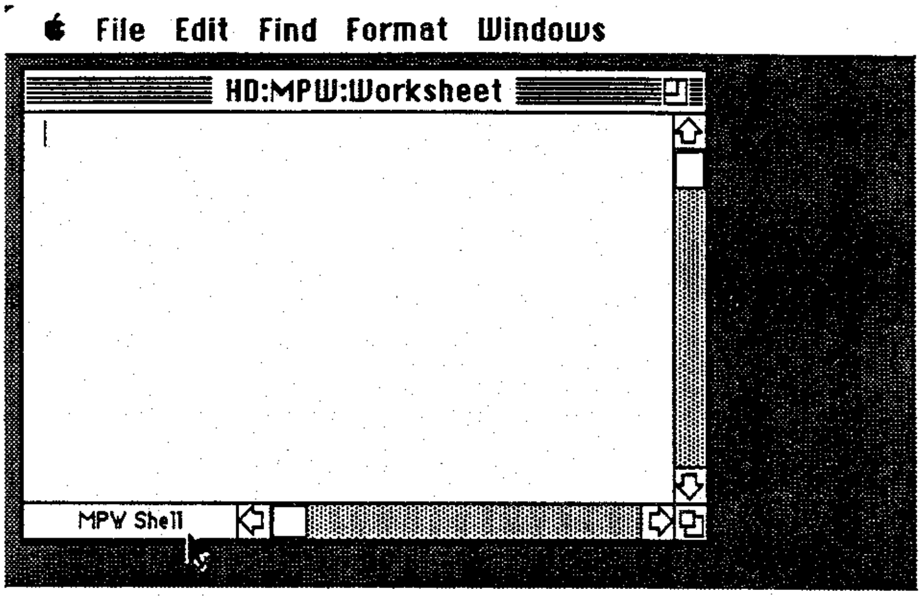


Figure 1-1
Worksheet window

A **status panel** at the window's lower-left corner shows the name of the command that's currently executing. When you first start the Macintosh Workshop, it begins by executing a command file called Startup. The **Startup file** defines several variables and command aliases (alternate command names); this file is further described in Chapter 3.

Important

The Startup file must be in the same directory as the MPW Shell, or in the System Folder.

Editing

Basic editing functions are available as menu commands. You can open a file with the open command or by selectiong it's name on the screen and choosing the open selection command (command-W) 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 fully described in Chapter 2, "Basic Editing."

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 topmost window), but they act on text in the **target window** (the second window from the top), or another window that you explicitly name. The command language lets you produce command files 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 2.

Giving commands

In MPW, **commands** may be either built-in commands, tools, applications, or command files, as explained in the "System Overview" section. Commands are written as a series of words separated by spaces or tabs. By default, command output and any error messages appear immediately after the command. 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 lists the date and time:

```
Tuesday, February 14, 1987 7:12:00 AM
```

Commands can have parameters. For example:

```
Date -d
```

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

```
Tuesday, February 14, 1987
```

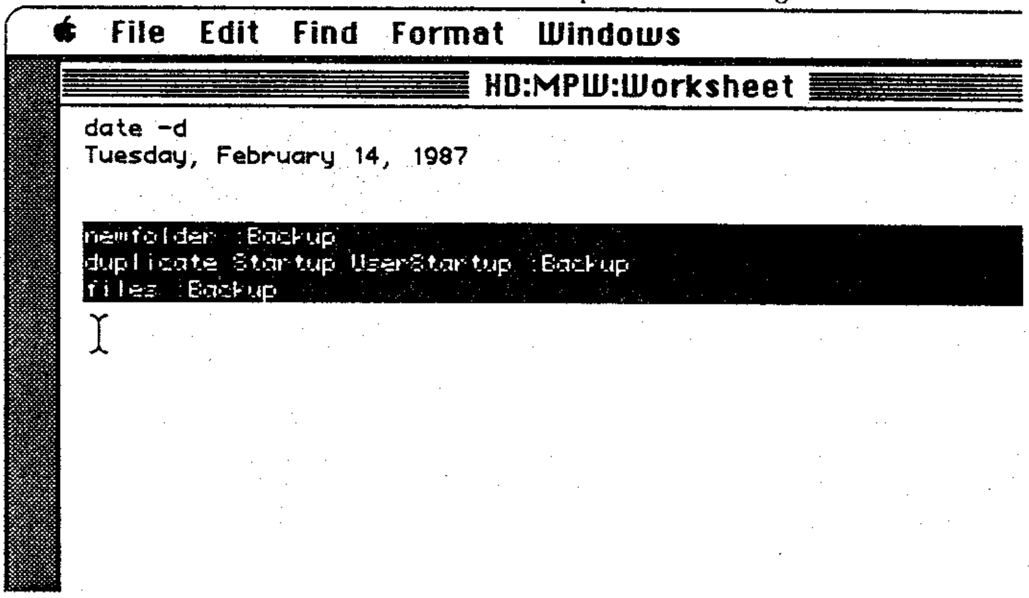
The Enter key

The **Enter key** serves as a "do it" button, causing commands to be executed. You can type commands from the keyboard and hit Enter to execute the command line. (When no text is selected, the entire line is executed, regardless of where the insertion point is on the line.) You can also select command text that is already on the screen and press the Enter key to execute the selected text.

The Enter command on the Edit menu has the same effect as pressing the Enter key. Command-Return is also equivalent to Enter.

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 1-2.



```
date -d
Tuesday, February 14, 1987

newfolder :Backup
duplicate Startup UserStartup :Backup
files :Backup
```

Figure 1-2
Press Enter to execute selected text

In Figure 1-2, executing the selected text would first make a new folder (directory) named Backup; then copy the files Startup into Backup, [redacted]; and then list all of the files in Backup. (Each of these commands, and the pathname syntax, is described in the following sections.)

You can also directly execute text files that contain other commands, simply by entering the filename of the command file. Executing a command file 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 3).

Important

Commands that don't produce any output run silently; this facilitates their use in command files.

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 ("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 3.)

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
    -l                  # long format (type, creator, size, dates, etc.)
    -q                  # don't quote filenames with special characters
    -r                  # recursively list subdirectories
    -t type             # list only files with this type
```

❖ *Note:* The square brackets are a syntax element indicating that a parameter is optional. Ellipses (...) indicate that the preceding item may be repeated. Syntax notation is described in the Preface to this manual. The number sign (#) is the MPW comment character.

You can directly edit and execute the text on the screen. For example, you can edit the above text as follows:

- use the mouse to select `[option...]` and `[name...]`; replace them with the option **`-l`** and the directory name AExamples
- remove the output specification > **`fileList`**

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

```
Files -l AExamples
```

(AExamples is the directory containing sample assembly-language programs; -l is an option that generates "long" output.) Press Enter to execute the command—directory information will appear immediately following the command.

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

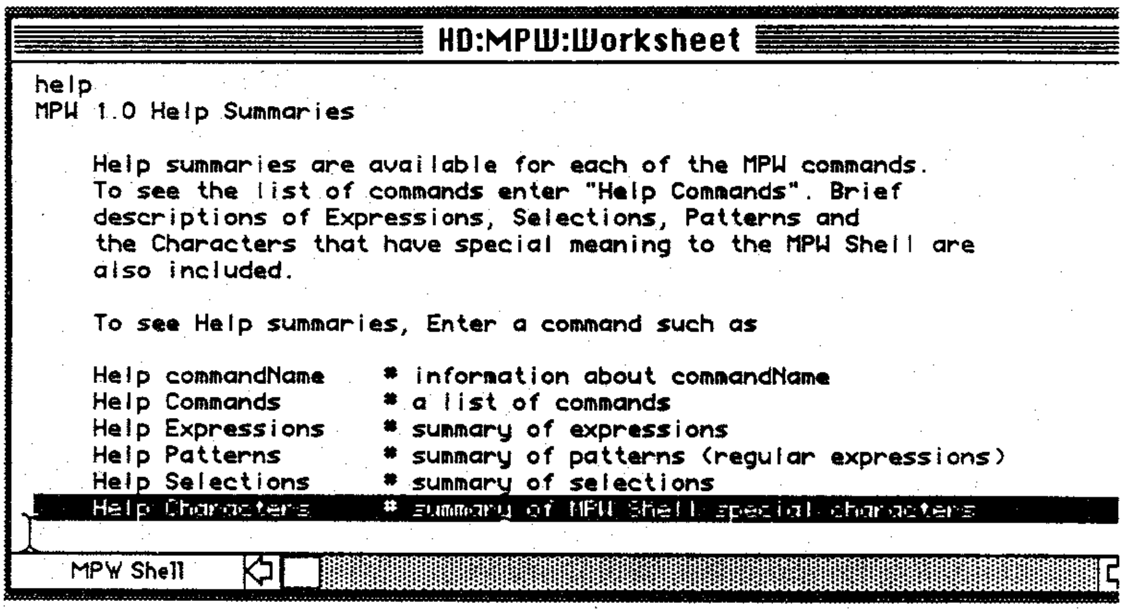
- an annotated list of all MPW commands
- an annotated list of the characters that have special meanings to the MPW Shell
- descriptions of the syntax of expressions, selections, and text patterns

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

```
Help
```

This command displays the information shown in Figure 1-3

Figure 1-3
Help summaries



```
help
MPW 1.0 Help Summaries

    Help summaries are available for each of the MPW commands.
    To see the list of commands enter "Help Commands". Brief
    descriptions of Expressions, Selections, Patterns and
    the Characters that have special meaning to the MPW Shell 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
```

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

File handling commands

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

❖ *Note:* The descriptions in the table omit some of the command options that are available. For complete descriptions, see Chapter 9, "Command Reference."

Table 1-1
Basic file handling commands

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.)
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	Directory with no parameters writes the pathname of the current directory.

Duplicate *name*... *targetName*	Duplicate file or directory *name* to file or directory *targetName*.
Files [*name*...]	List names of directories and files.
Move *name*... *targetName*	Move file or directory *name* to *targetName*.
New [*name*]	Open a new window.
NewFolder *name*...	Create the new directory *name*.
Open *name*	Open a window.
Rename *name1* *name2*	Rename *name1* to *name2*.
Save *window*	Save a window.
Volumes [*name*...]	List mounted volumes.
Windows	List open windows.

File and window names

❖ *Note:* All of the examples in this manual assume a hard disk named "HD:", using the hierarchical file system. If you are using a non-HFS system, all pathname specifications should be omitted.

In the Macintosh Workshop, 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 wisest 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 3.

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 possibilities:

§	Currently selected text in the target window. (The **target window** is the second window from the top, as explained in Chapter 2.)
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 2.

❖ *Note:* The MPW Shell uses a number of special metacharacters (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. A **full pathname** is specified as follows:

volume-name:[*directory-name*:...]*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.)

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. To maintain compatibility with the nonhierarchical file system, the following definition is applied: ***Any name that contains no colons or begins with a colon is considered a partial pathname.***

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 "File Manager" chapter 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.

Figure 1-4 shows a directory "tree" describing your MPW files.

Figure 1-4
Hierarchical directory structure

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:

```
:AExamples:
:AIncludes:
:Applications:
:Debuggers:
:Libraries:
'MPW Shell'
MPW.Help
Quit
Resume
:RIncludes:
Startup
Suspend
SysErrs.Err
:Tools:
UserStartup
Worksheet
```
etc.

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 partial pathname), the Shell searches for the command in the directories listed in the Shell variable {Commands}. This search path is initially set to: (the current directory), MPW: Tools, and MPW: Applications, as described in Chapter 3.

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 AExamples in the MPW folder with the command

```
Directory HD:MPW:AExamples
```

Like most commands, Directory runs silently—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 default directory.

Remember that to specify a pathname containing spaces or other special characters, you'll need to quote it by surrounding it with sigle or double quotes.
(See Chapter 3.)

An aside: the Alias command

For frequently used commands such as Directory, you may get tired of typing the entire command name. You can easily define your own alternate names with the Alias command. For example:

```
Alias CD Directory
```

After executing this command, you can execute the Directory command by entering the new command name:

```
CD
```

To make an alias definition part of the Shell's standard startup procedure, place the definition in the file UserStartup.

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:". Thus, the previous Directory command could be entered as

```
Directory "{MPW}AExamples"
```

In this particular case, the quotes aren't necessary, but their use is recommended as a general practice when variables are included in a pathname, because the pathname could contain spaces or other special characters.

You can use the Set command to define and redefine variables, as described in Chapter 3. 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 3.)

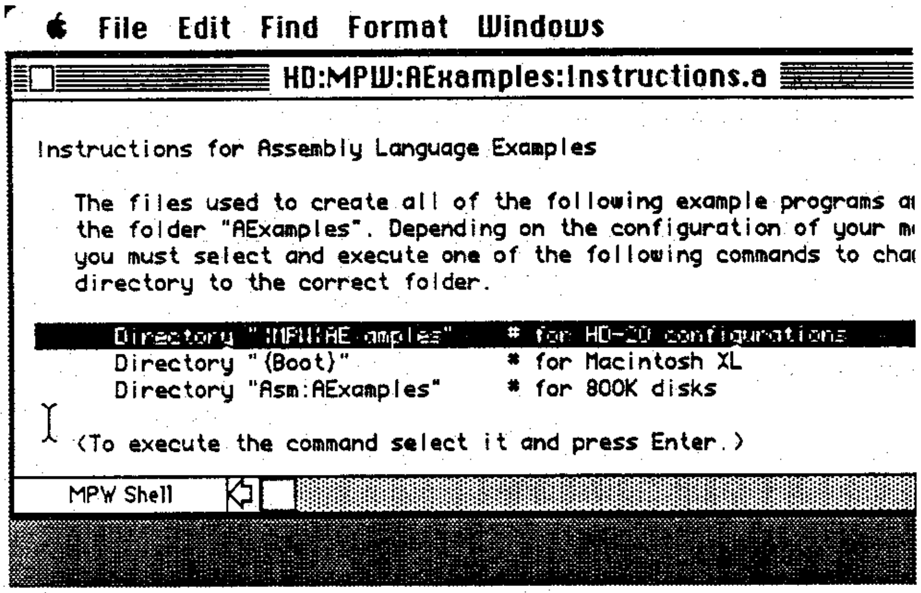
Building a program

This section introduces the MPW tools for assembling or compiling and linking a program, using the Sample application provided with MPW. (This application is the same as the *Inside Macintosh* sample application, and is available in Pascal, C, and assembly language.) Instructions files are provided for each language. To see these instructions for assembly language, open the file named Instructions.a in the AExamples folder. You can select and enter commands directly from the Instructions file, just as you can from any text file.

❖ *Note:* The examples in this section refer to assembly language, but the process is the same for C or Pascal.

Assembling and compiling

Before executing the example assemble or compile commands, you'll need to set the appropriate default directory (AExamples, PExamples, or CExamples) by selecting and entering the proper Directory command, as shown in Figure 1-5.



```
Instructions for Assembly Language Examples

   The files used to create all of the following example programs a
   the folder "AExamples". Depending on the configuration of your m
   you must select and execute one of the following commands to cha
   directory to the correct folder.

      Directory ":MPW:AExamples"     # for HD-20 configurations
      Directory "(Boot)"             # for Macintosh XL
      Directory "Asm:AExamples"      # for 800K disks

   (To execute the command select it and press Enter.)
```

Figure 1-5
Setting the default directory

❖ *Note:* The number sign (#) is the MPW comment character.

You can assemble or compile the Sample program with one of these commands:

```
Asm Sample.a

Pascal Sample.p

C Sample.c
```

Unless errors occur, the Assembler and Compilers run silently, like most MPW commands. The spinning "beachball" cursor indicates that the command is executing. (The name of the currently executing command is shown in the status panel at the active window's lower-left corner.)

❖ *Note:* Asm, Pascal, and C are not built-in commands: they are separate files on the disk. If you've installed your MPW files as specified at the beginning of this chapter and set the appropriate default directory, you won't notice any difference. Otherwise, you may have to specify the command's full pathname. (The Shell expects to find Asm, Pascal, and C in the Tools folder; the appropriate pathname was defined in the Startup file.)

If the command returns with an error message indicating that it wasn't found, check the installation instructions at the beginning of this chapter. Make sure that the Startup file is in the same folder as the Shell application, and that the other files and folders are set up as specified; then restart MPW.

The Asm command produces the object file Sample.a.o. To create an executable file, this file must be linked, and combined with the additional resources needed by the program.

Compiling resources

In addition to code, the Sample application includes a number of other resources (a window, menus, and so on). A textual description of these resources is contained in the file Sample.r. You can use the Resource Compiler, Rez, to compile these resources as follows:

```
Rez Sample.r -o Sample
```

This command compiles the resources described in Sample.r, placing them in the resource fork of the file Sample. (**-o** is a Rez option for specifying the output file.)

Note that you can also use the interactive resource editor, ResEdit, to create or modify resource files. See Chapter 5 for information.

Linking

The Linker links object code and produces executable code resources; these resources are placed in the resource fork of the output (**-o**) file, without disturbing other resources in the file. To link the sample assembly-language program, enter the command

```
Link Sample.a.o -o Sample
```

This command produces the executable application Sample. For the Pascal or C programs, you'll also need to link your object code with a number of library files. A shortcut is described in the next section.

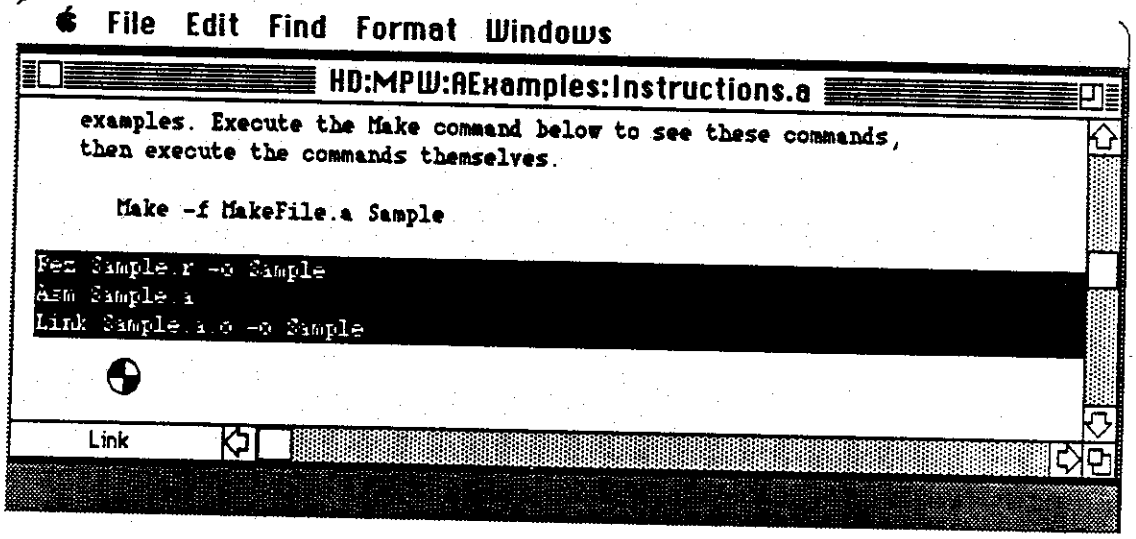
Automating the program build process with Make

When a program has more than a few modules, it becomes difficult to keep track of which other parts of a program need to be recompiled after you update a particular module. The Make command helps you keep track of these dependencies—it does this by referring to a text file called a **makefile**, which contains a set of dependency rules. A sample makefile has been provided in each of the Example folders.

To see how Make works with the assembly-language sample program, refer again to the Instructions file in the AExamples folder. To automatically generate the commands for building Sample.a, enter the command

```
Make -f MakeFile.a Sample
```

If the Sample program is up-to-date with respect to its component files (Sample.a and Sample.r), Make will generate no output. If Sample needs to be updated (or does not exist), Make will generate the set of commands required to rebuild it. This list of commands appears immediately after the Make command line. To build the file, just use the mouse to select these commands, as shown in Figure 1-6, and press the Enter key.



```
examples. Execute the Make command below to see these commands,
then execute the commands themselves.

    Make -f MakeFile.a Sample

Rez Sample.r -o Sample
Asm Sample.a
Link Sample.a.o -o Sample
```

Figure 1-6
Executing Make output for Sample

❖ *Note:* The ∂ character that appears in the Link command is the **Shell escape character**, and ∂Return functions as a line continuation character—see "Quoting Special Characters" in Chapter 3.

The result of these commands is an executable application, Sample.

Running an Application

You can run an application, like any other command, by selecting the application name (in this case, Sample) and pressing the Enter key. (The Sample application simply puts up a window and allows you to edit text in the window.)

You can also pass parameters to applications, as explained in Chapter 3. When you quit from an application, you'll return to the MPW Shell. Two special command files called Suspend and Resume save and restore Shell variables and other information when you run an application and return to the Shell.

Chapter 2

Basic Editing

Features xx

File format xx

Menu commands xx

Apple menu x
File menu x
Edit menu x
Find menu x
    Selection expressions x
        Selection by line numbers x
        "Wildcard" operators x
Format menu x
Windows menu x
User-defined menus x

Editing with the command language

This chapter describes simple editing using menu commands. Advanced editing capabilities are discussed in Chapter 4, "Advanced Editing."

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 on any of the characters

  ( )   [ ]   { }   "   '

  to select everything between the character and its mate. (To select text between quotes, click on the *left* quote.)
- 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 command files that define new editing commands. (See Chapter 4.)
- 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 4.)

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, regardless of the file creator.

❖ *Note:* 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) with the Show Invisibles menu item.

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

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.)

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

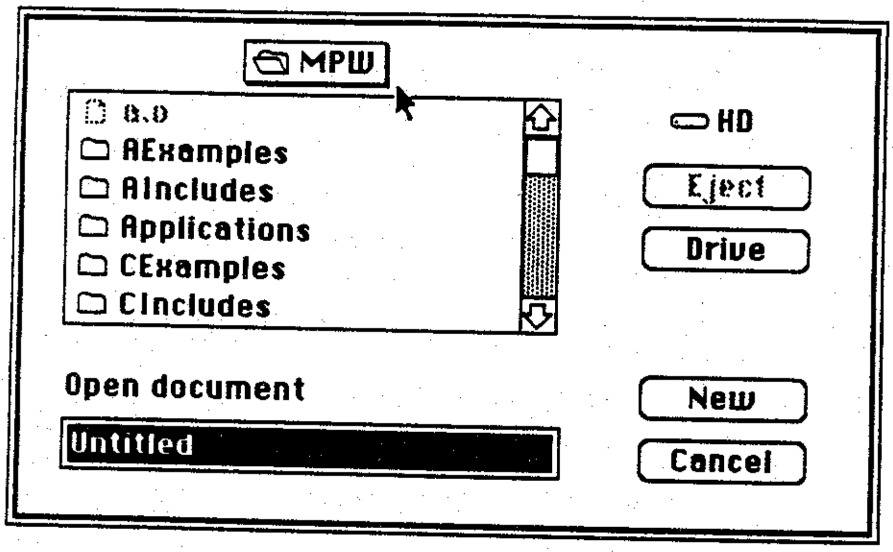
❖ *Note:* Many menu items (including several items in the File menu and all user-defined items) are disabled when running commands. This prevents you from closing windows that the command may be reading, and from trying to run another command at the same time.

Menu

About the MPW Shell	Displays version and copyright information.

File Menu

New...	Puts up a New dialog, shown below. The MPW New dialog allows you to enter a name and select a directory location for the document.



Open...	Puts up an Open dialog 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. For subsequent Opens, the file 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. *Note:* If you try to open a document that's already open in another window, that window will be brought to the front.
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. Variable and command substitution do not occur on the selection.
Close	Closes the active (topmost) window.
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.

Save as…	Puts up a Save As dialog, 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.
Page Setup…	Puts up the standard Page Setup dialog.

Print Window/
Print Selection
	Prints either the entire 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 contents of the entire window are printed.
	
	The Print menu item doesn't put up the usual Print dialog. Instead, you can specify printing parameters by setting the Shell variable {PrintOptions}, described in Chapter 3. Printing options include the number of copies to print, which pages to print, print quality, font and font size, headings and title, borders, and printing the pages in reverse order (for use with the LaserWriter). See the description of the Print command in Chapter 9 for a complete specification of these options, or enter the command Help Print to see a summary.
	
	Technical Note: 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.
	
	Important: For the Print command to work properly, you must install the printer drivers available on version 1.0 or later 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 to specify paper size, orientation, and reductions or enlargements.

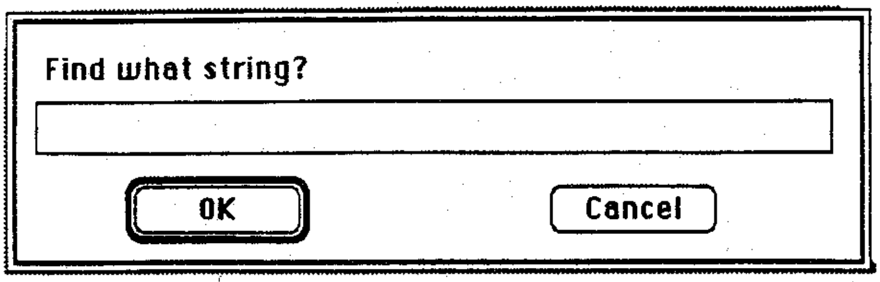
Quit	Returns to the Finder, first allowing you to save the current state of all open files.

Edit menu

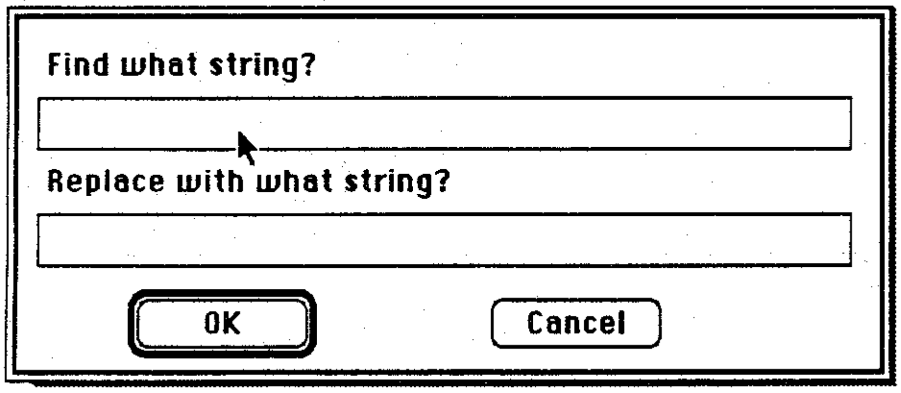
Undo	Undoes the most recent changes to *text* in the active window (but *not* changes to resources such as the cursor position or font and tab settings). You can select Undo again to redo changes.
Cut	Copies the current selection in the active window to the Clipboard, and then deletes it from its original location.
Copy	Copies the current selection in the active window to the Clipboard.
Paste	Replaces the contents of the current selection in the active window with the contents of the Clipboard.
Clear	Deletes the current selection in the active window.
Select All	Selects the entire contents of the active 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 adds a tab at the beginning of each line. Shift Right removes a tab, or the equivalent number of spaces, from the beginning of each line. 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.
Enter	Executes the currently selected text. This is exactly the same as pressing the Enter key.

Find menu

Find...	Puts up a Find dialog, and finds the string you specify. By default, the editor searches forward from the current selection in the active window (and does not wrap around).



Find Same	Repeats the last Find operation, on the active window.
Find *selection*	Finds the next occurrence of the current selection in the active window.
Display Selection	Scrolls the current selection in the active window into view.
Replace...	Puts up a Find-and-Replace dialog:



Replace Same	Repeats the last Replace operation.

❖ *Note:* For Find and Replace operations, a beep indicates that the selection was not found.

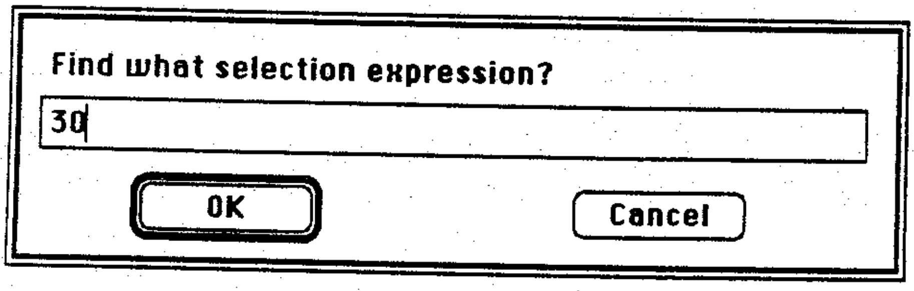
Four switches govern the operation of the Find and Replace commands. (A check mark indicates that an item is selected.)

Search Backward	Search backward, from the current selection to the beginning of the file. (Normally, searching is forward, and stops at the end of the file.)
Entire Word	Search for entire words only. To the editor, a word is composed of the characters a–z, A–Z, 0–9, and the underscore character (_). (These default values can be changed by redefining the Shell variable {WordSet}—see "Predefined Variables" in Chapter 3.)
Case Sensitive	Searching is normally case insensitive; selecting this menu item specifies case-sensitive searching. (It does this by setting the Shell variable {CaseSensitive}—see "Variables Defined in the Startup File" in Chapter 3.)
Selection Expression	Enables the full selection and regular expression syntax, as used with the command language and described in Chapter 4. These expressions allow powerful selection and pattern matching capabilities that use a special set of metacharacters, introduced below.

Selection expressions

When the Find menu'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, and a full discussion is deferred to Chapter 4, "Advanced Editing."

Selection by line number. A number given by itself specifies a line number. For example:


Find what selection expression?

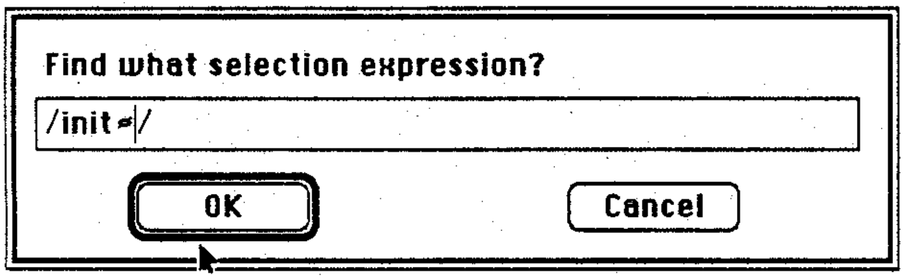
30

This command selects line number 30 in the active window.

"Wildcard" Operators. The same wildcard operators used in filename generation can also be used to specify text patterns for Find commands:

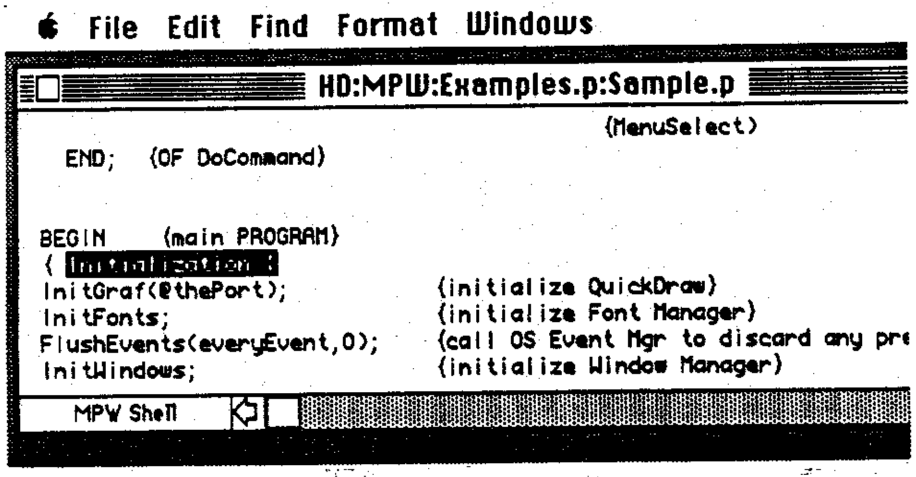
?	Any single character (other than Return).
≈	Any string of 0 or more characters, not containing a Return (to get the ≈ character, press Option-X).
[*characterList*]	Any character in the list. *Note:* The brackets must be typed; they don't indicate an optional syntax element.
[¬*characterList*]	Any character ***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**. A regular expression consists of literal characters and/or regular expression operators, and must be enclosed in slashes (/.../). For example:


Find what selection expression?

/init≈/

This command finds and selects any string that begins with "init", and is followed by any characters other than a return, as shown in Figure 2-1.



```
                                        {MenuSelect}
  END;  {OF DoCommand}


BEGIN    {main PROGRAM}
{ Initialization }
InitGraf(@thePort);               {initialize QuickDraw}
InitFonts;                        {initialize Font Manager}
FlushEvents(everyEvent,0);        {call OS Event Mgr to discard any pre
InitWindows;                      {initialize Window Manager}
```

Figure 2-1
Text selected with the find command

As mentioned, many additional Find and Replace capabilities are available—see Chapter 4. In the command language, the Find menu functions are performed by the Find and Replace commands. There's also a tool named Search that can search through a list of files for the occurrence of any text pattern.

Format menu

Tabs...	Puts up a dialog that lets you set the number of spaces that a tab character will signify for the active window. (The default Tab setting is set by the Shell variable {Tab}, described in Chapter 3.)
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.)
Show Invisibles	Displays the invisible characters as follows:

Tab	Δ
Space	◊
Return	¬

9 Point	
10 Point	
etc.	
Chicago	
Courier	
etc.	The rest of the menu consists of a selection of the fonts installed in your System file. Available font sizes are outlined. *Note:* Selecting a font and font size affects the ***entire*** active window, not just the current selection in that window.

Windows menu

The Windows menu lists all open windows. Selecting a window from the menu brings that window to the top. The name of the target window is underlined. A check indicates that a window contains changes that have not yet been saved.

Clipboard	The Clipboard is always listed first, and appears even if the Clipboard isn't open. You can therefore use this menu item to open the Clipboard. The pathname of the Clipboard includes the directory that contains the MPW Shell.
Worksheet	The Worksheet always appears second in the Windows menu. The menu item lists the full pathname of the Worksheet.

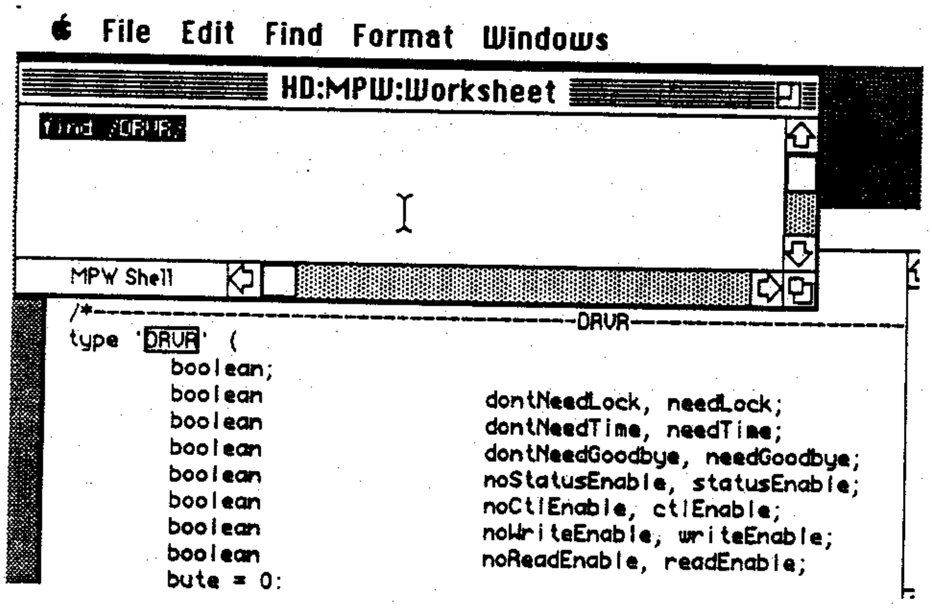
User-defined menus

You can define your own menu commands with the AddMenu command, described in Chapter 3. These commands can be appended to existing menus, or you can create new menus.

Editing with the command language

Almost all menu functions have equivalents in the command language. In most respects, there is no difference between the menu items and their command-language equivalents. The primary difference is that with the command language, you enter commands in the active (topmost) window, and the 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** (the second window from the top).

For example, to use command-language techniques to edit the file SysTypes.r, you must first open that file, and then click on another window, such as the Worksheet window, to make it the active window. You'll enter your commands in the active window, as shown in Figure 2-2. 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 2-2.



```
find /DRVR/
```

```
/*-----------------------------------DRVR-------------------
type 'DRVR' (
        boolean;
        boolean                     dontNeedLock, needLock;
        boolean                     dontNeedTime, needTime;
        boolean                     dontNeedGoodbye, needGoodbye;
        boolean                     noStatusEnable, statusEnable;
        boolean                     noCtlEnable, ctlEnable;
        boolean                     noWriteEnable, writeEnable;
        boolean                     noReadEnable, readEnable;
        byte = 0:
```

Figure 2-2
Text highlighted in the active window and target window

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 indicates 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 4. The Clear command given above is so useful that you may want to add it as a menu item—see Chapter 3 for a description of adding your own menu items to MPW.

Chapter 3

Using the Command Language

Overview xx

Types of commands xx

Entering and executing commands xx

Structure of a command xx

Command name x
Parameters x
Command terminators x
Comments x
Simple versus structured commands x

Running an application outside the Shell environment xx

Command files xx

Special command files xx

The Startup and UserStartup files x
Suspend, Resume, and Quit x

Command aliases xx

Executable error messages x

Variables xx

Predefined variables x
Variables defined in the Startup file x
Parameters to command files x

Defining and redefining variables x
Exporting variables x

Command substitution xx

Quoting special characters xx

How commands are interpreted xx

Structured commands xx

Control loops x
Processing command parameters x
Expressions x

Filename generation xx

Redirecting input and output xx

Standard input x
    Terminating input with Command-enter x
Standard output x
Diagnostic output x
Pseudo-filenames x

Defining your own menu commands xx

Sample command files xx

"AddMenuAsGroup" x
"CC" x

So far, we've introduced only isolated groups of commands without treating the Shell's command language as a whole. This chapter describes the complete syntax of the MPW command language and explains its use. The commands themselves are described in Chapter 9, "Command Reference."

Overview

Besides the built-in commands already introduced, the command language provides the following features:

- built-in and user-definable **variables** of the form {*variableName*}
- command **aliases,** used to create alternate names for commands
- **command substitution,** by which commands enclosed in back quotes, `…`, 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 <, >, >>, ≥, and ≥≥ 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 of this chapter sections as 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 Files" and "Special Command Files"
- "Variables"
- "Quoting Special Characters"

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

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.
- **Tools**, such as Link or Asm, are executable programs (that is, separate files on the disk) that are fully integrated with the Shell environment.
- **Command files**, 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 command file and use them in commands within the file.
- **Applications**, such as ResEdit or MacPaint™, are stand-alone programs that can be launched from the Shell, but run outside the Shell environment.

To execute a tool, application, or command file, the proper program file needs to be on your disk.

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

Entering and executing commands

Pressing the Enter key executes command text. You can select command text on the screen and press the Enter key to execute the selected text. If no text is selected, pressing Enter executes the entire line that contains the insertion point.

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).

Caution

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

The Enter menu item and the key combination Command-Return both have the same effect as the Enter key.

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

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 command file to execute. The command name is passed as parameter 0, and can be referenced by command files in the variable {0}, explained below under "Variables." Command names are not case sensitive. Alternate names can be defined for a command—see "Command Aliases" in this chapter for information.

Parameters

Each of the subsequent words in a command is a **parameter** to the command or to the Shell. You can reference parameters within command files by using the variables {1}, {2},... {*n*}. (See Table 3-4.) Note that certain parameters, such as I/O redirection, are interpreted by the Shell, and never seen by the program. 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 Chapter 9 for more details on these conventions.

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 3-1 describes the command terminators in order of precedence.

Table 3-1
Command Terminators

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 execute sequentially.
cmd1 **&&** *cmd2*	Executes *cmd2* only if *cmd1* succeeds (that is, returns a status value of 0).
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. They all group from left to right. Parentheses can be used to group commands for conditional execution and pipe specifications. Some examples follow.

```
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.

❖ *Note:* 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. (For more information about the ∂ escape character see "Quoting Special Characters" in this chapter.)

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

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 ∂.)

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, which let you control the order in which other commands are executed. All of the structured commands are built-in, and usually have more than one keyword. The entire structured command is read before its execution begins. For example,

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

For information, 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 Segment Loader chapter 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 the directory MPW:Applications:), 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.)

Caution

Running an application from a command file terminates the command file.

Command files

You can create your own commands by writing text files of previously defined commands. 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 the Enter key. You could also save this text as a command file so that it's always available. To save it under the name "Info", for example, you could first select the command text, and type the following command in another window:

```
Duplicate § Info
```

You can now execute this series of commands by entering the command name Info.

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

filename [*parameters*...]

Parameters can be referred to within the command file by using the built-in variables `{1}`, `{2}`,...`{n}`, explained below under "Parameters to Command Files."

❖ *Note:* As a matter of convenience, command files (as well as applications and tools) are usually kept in directories that the Shell automatically searches when a partial pathname is given for a command name. This convention allows you to invoke the command by using its simple name 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 command files

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

Important

Each of these files 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 command file 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 add-on menus. Aliases and variables are fully described in the following sections.

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, and restore the state after returning to the Shell.

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 command files run as if you had entered the commands interactively. You can modify them to suit any special requirements you might have.

Command aliases

An **alias** is an alternate 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 (a command may consist of more than one word).
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 command file in which they are defined (and are globally available if they are defined in the Startup file). Aliases are automatically inherited from enclosing command files, and may be redefined locally. However, aliases redefined locally will revert to their previous value when the command file terminates.

See the Alias and Unalias commands in Chapter 9 for a complete specification of aliases and additional 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 "Editing With the Command Language" in Chapter 2.) This alias is useful when a compiler returns an error message such as

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

By selecting the entire line and hitting 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 53
```

("Line" is a command file, which 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
- providing status information
- local variables in command files
- parameters to command files and tools
- setting 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,

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

This command 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. For example,

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

In this example, {PFiles} is replaced with its definition before the command is executed.

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

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

Predefined variables

Table 3-2 lists the variables defined by the MPW Shell. These variables provide the status value returned by the last command, and the pathnames of several files and directories.

Table 3-2
Variables defined by the 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.)
{Boot}	Volume name of the boot disk.
{Active}	Full pathname of the current active window.
{Target}	Full pathname of the target window (that is, the second window from the top—by default, this is the window where editing commands take effect).
{Worksheet}	Full pathname of the Worksheet window.
{SystemFolder}	Full pathname of the directory that contains the System and Finder files.
{ShellDirectory}	Full pathname of the directory that contains the MPW Shell.
{Command}	Full pathname of the last command executed. (For built-in commands, this is the name of the command.)

Variables defined in the Startup file

Table 3-3 lists the variables that are defined in the Startup file (described in the "Special Command Files" section in this chapter). These variables define pathnames and default settings to the Shell, and are referenced by the Shell and by some of the Workshop tools. You can change any of these definitions to suit your own needs.

❖ *Note:* Hierarchical file system (HFS) pathname conventions are described in Chapter 1.

Table 3-3
Variables defined in the Startup file

Variables referenced by the command interpreter:	
{MPW}	The volume or folder containing the Macintosh Programmer's Workshop. Initially set to "{Boot}MPW:".

{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}Applications:` —that is, the current directory, then HD:MPW:Tools, and then HD:MPW:Applications.
{Exit}	When {Exit} is set to a nonzero value, command files terminate whenever a command returns a nonzero status. This nonzero status is returned as the status value of the command file. (See the {Status} variable in Table 3-2.) {Exit} is initially set to 1.
{Echo}	When {Echo} is set to a nonzero value, commands are written to diagnostic output after variable substitution, command substitution, and filename generation, and just prior to execution. This capability is useful for watching the progress of a command file and for debugging command files—as the first line of your file, you would include the line `Set Echo 1` {Echo} is initially set to 0.
{Test}	When {Test} is set to a nonzero value, the command interpreter executes built-in commands and command files, 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.

Variables referenced by the editor:

{CaseSensitive}	Any nonzero value specifies case-sensitive pattern matching. {CaseSensitive} is initially set to 0 (that is, false). You can also set {CaseSensitive} by selecting the "Case Sensitive" item from the Find menu. (See Chapter 2.)
{Tab}	Default tab setting for new windows (initially 4).
{WordSet}	The set of characters that constitute a word to the editor (for 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.
{PrintOptions}	Options used by the Print Window and Print Selection menu commands. Initially set to `'-h'`. (The **-h** option prints pages with headers. For more information on possible print options, see the Print command in Chapter 9.)

Pathnames for libraries and include files:

{RIncludes}	The directory that contains Resource Compiler (Rez) include files. Initially set to `"{MPW}RIncludes:"`.

{AIncludes}	The directories to search for assembly-language include files, referenced by the Assembler. Initially set to `"{MPW}AIncludes:"`.
{Libraries}	The directory that contains shared library files. Initially set to `"{MPW}Libraries:"`.
{CIncludes}	The directories to search for C include files, referenced by the C Compiler. Initially set to `"{MPW}CIncludes:"`.
{CLibraries}	The directory that contains C library files. Initially set to `"{MPW}CLibraries:"`.
{PInterfaces}	The directories to search for Pascal interface files, referenced by the Pascal Compiler. Initially set to `"{MPW}PInterfaces:"`.
{PLibraries}	The directory that contains Pascal library files. Initially set to `"{MPW}PLibraries:"`.

❖ *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 as follows:

```
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.")

Parameters to command files

When a command file is executed, its parameters automatically set the value of certain Shell variables. These variables are explained in Table 3-4.

Table 3-4
Parameters to command files

{0}	Name of the currently executing command file.
{1}, {2},...{*n*}	First, second, (or *n*th) parameter passed to the current command file. (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 command file (which is useful with error messages as explained above 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 command file 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 quotes 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's a filename that might contains 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.)

Caution

Removing all variables in the outermost scope can have serious consequences. For example, the Shell uses the variable {Commands} to locate MPW tools, and 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 command files and programs involves two separate steps:

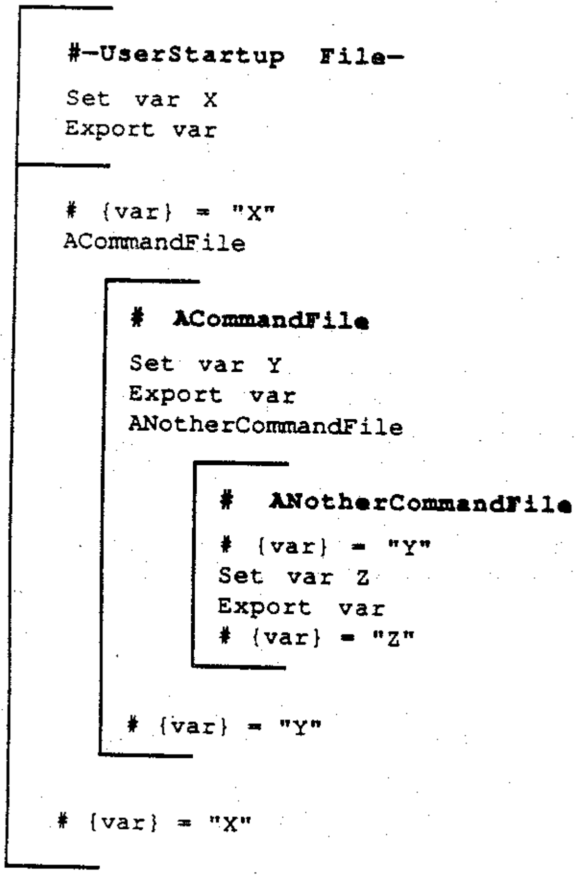
1. You can define a variable with the **Set** command. Note that variables are local to the command file in which they are defined—a variable definition ceases to exist when its command file terminates.
2. You can pass a variable to command files and tools with the **Export** command. After you export a variable, nested command files can reference that variable, and may override its value locally—but any redefinition is strictly local, and terminates when the command file terminates. It's impossible to affect the value of a variable in an enclosing command file. (See Figure 3-1.)

Exporting variables

The Export command makes variables available to command files and tools:

Export *name...*	Exports the named variables.
Export	Writes the list of exported variables to standard output.

You can define a variable globally by setting its value in the Startup file and exporting it. Figure 3-1 illustrates how Export works.



```
#-UserStartup  File-
Set  var  X
Export var

# {var} = "X"
ACommandFile

    #  ACommandFile
    Set  var  Y
    Export  var
    ANotherCommandFile

        #  ANotherCommandFile
        # {var} = "Y"
        Set  var  Z
        Export  var
        # {var} = "Z"

    # {var} = "Y"

# {var} = "X"
```

Figure 3-1
Trafficking in variables

Startup can be thought of as the command file enclosing all other commands (including interactive commands).

❖ *Note:* You can use the Execute command to execute a command file without creating a new scope for variables, exports, and aliases. The Shell "executes" the Startup, Suspend, Resume, and Quit command files, and Startup uses Execute to run the UserStartup script. For more details about Execute, see Chapter 9.

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 back quotes, `...` (`/~ ).. 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" quotes (that is, the standard single quotes '...').

❖ *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, 1986 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"
```

`` `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, which you can't do with wildcard operators.

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.

Table 3-5 lists all of the special symbols recognized by the Shell.

Table 3-5
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 3-1)
|	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	
#	Comments	"Structure of a Command"
∂	Escape character: quotes the following character	this section
'...'	Quotes all other special characters	(Table 3-6)
"..."	Quotes other special characters, except ∂, {, and `	
/.../	Quotes other special characters, except ∂, {, and `	
\\...\\	Quotes other special characters, except ∂, {, and `	
{...}	Variable substitution	"Variables"
\`...\`	Command substitution	"Command Substitution"
?	Matches any character in filename generation	"Filename Generation"
≈	Matches any string in filename generation	
[...]	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	
<	Input file specification	"Redirecting Input and Output"
>	Output file specification	(Table 3-10)
>>	Output file specification (append)	
≥	Diagnostic file specification	
≥≥	Diagnostic file specification (append)	
…	Reserved for future use	

Note: Within regular expressions (/.../ or \...\), a number of characters not listed here are also considered special. See "Pattern Matching" in Chapter 4 for details.

You can literalize a character by preceding it with the Shell escape character, ∂ (Option-D), or by including it within the quote symbols '...', "...", /.../, or \...\. The escape character, ∂, quotes a single character only; the other quote symbols may be used to quote part or all of a word. These symbols are described in Table 3-6.

Table 3-6
Quotes

∂*c*	Escape character: Take the single character *c* literally. ∂Return is discarded, allowing you to continue a command onto the next line. *Note:* The combinations ∂n, ∂t, and ∂f are exceptions to this pattern: they are used for inserting return, tab, and form feed characters, respectively.
'...'	"Hard quotes": Take the enclosed string literally—no substitutions occur. The quotes are removed before execution.
"..."	"Soft quotes": Take the enclosed string literally. ∂*c*, variable substitutions, and command substitutions occur. The quotes are removed before execution.
/.../ or \...\	Regular expression quotes: Normally used to enclose regular expressions. Take the entire string literally, including the quote characters—the / or \ characters are *not* removed. Variable substitutions and command substitutions occur. '...', "...", and ∂ have their usual meanings—however, they are not removed.

Single quotes, double quotes, and ∂ are removed before parameters are passed to programs (unless they are themselves enclosed in quotes). For example, here is how you could define an AddMenu that compiles a C program in the 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 quotes defeat variable expansion; they are then stripped off before the item is actually added. The double quotes 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 the "..." form of the 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 4 and the description of the Search command in Chapter 9.)

How commands are interpreted

When you send text to the 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, for instance.)
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.
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 the unquoted characters ?, ≈, [, *, +, or « 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.**

Any part of this process can be suppressed by using quotes as described in the previous section. Single and double quotes are removed prior to execution.

Structured commands

Structured commands (listed in Table 3-7) override the normal sequential execution of commands. They can be used interactively and within command files. They may be nested arbitrarily deeply (subject to a limitation on stack space). The entire structured command is read before execution begins.

Caution:

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).

❖ *Note:* Expressions (used in If, Break, Continue, and Exit) are defined in the section following the table.

Table 3-7
Structured commands

(*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...	`If` *expression*
	  command...
	`[Else If` *expression*
	  command...] ...
	`[Else`
	  command...]
	`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...

```
For name In word...
  command...
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. (See the examples below.)

Loop...End

```
Loop
  command...
End
```

Repeatedly executes the enclosed commands. The Break command is used to terminate the loop.

Break

```
Break [ If expression ]
```

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).

Continue

```
Continue [ If expression ]
```

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 ]
```

Exit terminates execution of the command file in which it appears. If *number* is present, it is returned as the status value of the command file; otherwise, the status of the last command executed is returned. If the expression is present, the command file 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 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 can't be produced as a result of variable substitution or command substitution.

You can apply conditional execution (&& and | |), pipe specifications (|), and input/output specifications (<, >, >>, ≥, and ≥≥) 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 word 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 Chapter 9.

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—you'll need to use the Break command to terminate the loop. You can break from a loop and continue with the next iteration with the Continue command. For example, the command file below runs a command several times, once for each parameter.

```
### Repeat - Repeat a command for several parameters ###
#
#     Repeat command parameter…
#
#      Execute 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 "{1}" == ""
      {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 command file Repeat, you could compile several C programs, with progress information, using the command

```
Repeat 'C -p' Sample.c Count.c Memory.c
```

Repeat might also be used to set the font and fontsize for all the open windows:

```
Repeat 'Font Courier 10' `Windows`
```

Processing command parameters

In addition to the commands introduced in Table 3-7, there are several other commands that are highly useful in command files. The following commands are used 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.
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, etc. If ***number*** is not specified, the default value is 1. Shift does not affect parameter {0} (the command name).

Echo and Parameters are useful for checking how your parameters will behave before actually passing them to a command (for instance, to check how your quotes are working out). For example

```
Parameters "≈"
```

For an example of how the various structured commands can work together, see "Sample Command Files" at the end of this chapter.

Expressions

Expressions are used in the If command and in If statements in the Break, Continue, and Exit commands. They're also used in the Evaluate command, which returns the result of an expression.

Table 3-8 lists the expression operators in order of decreasing precedence (some operators have several alternate symbols). Groupings indicate operators of the same precedence.

Table 3-8
Expression operators

	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
	>			Greater than
	>=	≥		Greater than or equal
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. Parentheses can be used 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 nonzero, non-null operands as true.

Numbers. Numbers may be either decimal or hexadecimal integers representable by a 32-bit signed value. Hexadecimal numbers begin with either **$** or **0x**. 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. For example

```
If {Status} != 0;  Beep -3a,25,200;  End
```

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-hand side is a regular expression against which the left-hand operand is matched. For example:

```
If "{1}" !~ /≈.[acp]/
   Echo Filename must end with .a, .c, or .p
End
```

❖ *Note:* The regular expression must be enclosed in the regular expression quotes, /.../. See Chapter 4, "Advanced Editing," 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 Chapter 9.)

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 (| | and &&)

- Pipe specifications (|)
- Input/output specifications (>, >>, ≥, ≥≥, and <)

This allows the use of many expression operators that would otherwise have to be quoted. For If commands, the conditional execution or I/O specification should come after the End word. For other commands that contains expressions, you can specify conditional execution or I/O redirection by enclosing the command in parentheses. For example,

```
(Evaluate {1} + {2}) ≥ Errors
```

Filename generation

After variables have been substituted, a 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 3-9.

Table 3-9
Filename generation operators

?	Matches any single character (except return or colon).
≈	Matches any string of zero or more characters (except return or colon).
[*characterList*]	Matches any character in the list.
[¬ *characterList*]	Matches any character not in the list.
*	0 or more repetitions (?* is the same as ≈)
+	1 or more repetitions
«...»	Numeric range

❖ *Note:* The pattern matching is case insensitive.

❖ *Note also:* The pathname separator (:) must appear explicitly in the pattern—the : character will never be substituted for ?, ≈, or [...].

These special characters are the same **regular expression operators** used in editing commands. For a complete discussion of regular expressions, see Chapter 4, "Advanced Editing."

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 is contained under the description of the Rename command in Chapter 9.

Redirecting input and output

All commands (built-in commands, command files, and tools) are provided with three open files: standard input, standard output, and diagnostic output (Figure 3-2). By default, standard input comes from the console; standard output and diagnostics go to the window where the command was executed, immediately following the command.

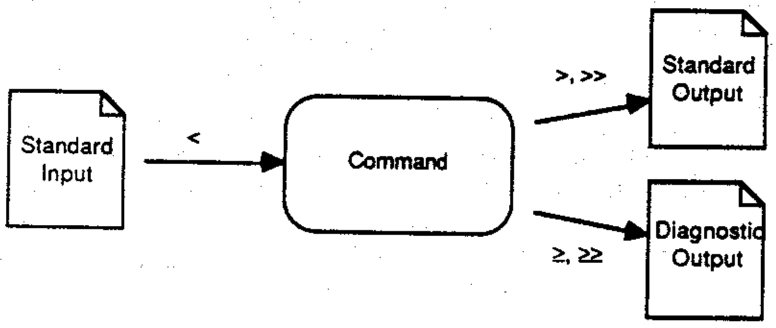


Figure 3-2
Standard input and output

You can override these default assignments with the <, >, and ≥ symbols described in Table 3-10. Note that input and output specifications are interpreted by the Shell; they are not passed to programs as parameters. Parentheses can be used to group commands for input/output specifications.

Table 3-10
I/O redirection

< *name*	Standard input is taken from ***name***.
> *name*	Standard output replaces the contents of ***name***. File ***name*** is created if it doesn't exist.
>> *name*	Standard output is appended to ***name***. File ***name*** is created if it doesn't exist.
≥ *name*	Diagnostic output replaces the contents of ***name***. File ***name*** is created if it doesn't exist.
≥≥ *name*	Diagnostic output is appended to ***name***. 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 tool running within the Shell environment, input always comes from the standard input file and output goes to the standard output file. The program doesn't need to know whether standard input happens to be text from a file, from a window, or 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 program—this process is completely invisible to the program.

I/O specifications also apply to command files. The standard input, standard output, and diagnostic output files provided to a command file become the defaults for commands in the file.

For more on input and output, see "Standard Input/Output Channels" in Appendix F.

Standard input

By default, standard input comes from the console. Normally, you supply this input 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
```

Therefore, the < operator is provided for completeness, and not because it provides significant new functionality. 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 it, and it 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. End-of-file is indicated by holding down the Command key and pressing Enter. 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. Command-Enter indicates end-of-file and terminates the command.

Standard output

By default, standard output appears in the active window 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,

```
Catenate File1 File2 > CombinedFile
```

The Catenate command concatenates File2 to File1—but instead of sending the output to the active window, it is sent to the file named CombinedFile. If window CombinedFile is open on the desktop, its contents are overwritten. Otherwise, file CombinedFile is replaced (or created if it doesn't exist).

The >> operator appends standard output to the end of a window or file. If the named file doesn't exist, a new file is created. For example,

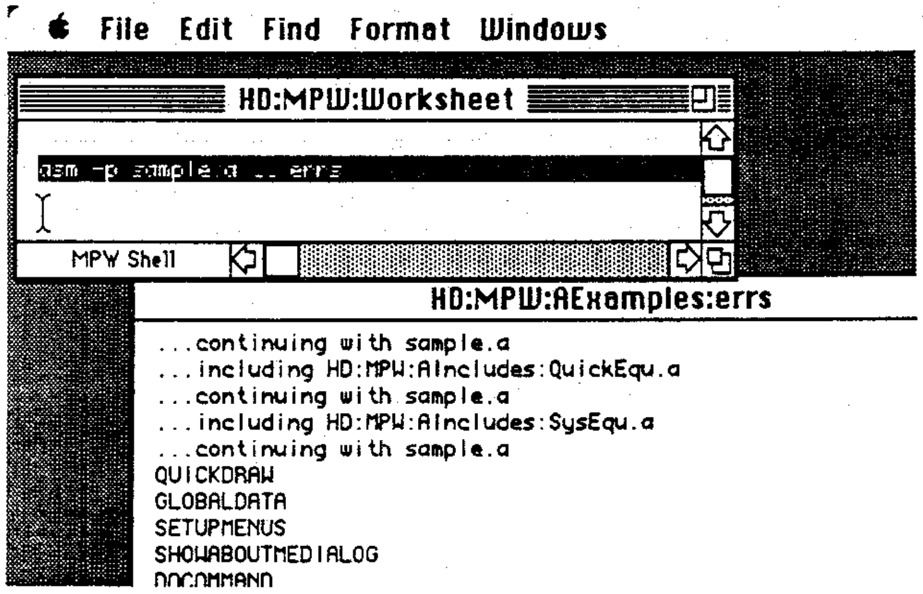
```
Catenate § >> AFile
```

appends the contents of the current selection 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 ≥ and ≥≥ 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 3-3, the Assembler has been run, and error and progress information has been appended to the Errs window.



```
asm -p sample.a ≥≥ errs
```

```
...continuing with sample.a
...including HD:MPW:AIncludes:QuickEqu.a
...continuing with sample.a
...including HD:MPW:AIncludes:SysEqu.a
...continuing with sample.a
QUICKDRAW
GLOBALDATA
SETUPMENUS
SHOWABOUTMEDIALOG
DOCOMMAND
```

Figure 3-3
Redirecting diagnostic output

Pseudo-filenames

Pseudo-filenames are a set of device names that you can use in place of filenames, but that 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 3-11 shows the available pseudo-filenames.

Table 3-11
Pseudo-filenames

Dev:Console	Always refers to the current console device. The console is the default source of input (that is, entered text) and the default destination of output (that is, the active window).
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	Default input stream.
Dev:StdOut	Default output stream.
Dev:StdErr	Default diagnostic output stream.

The last three names, StdIn, StdOut, and StdErr, are used to explicitly represent input and output. You can use these specifications, for example, to send a command's output and diagnostics to the same file:

```
Search /NULL/ ≈.c  > Found  ≥ Dev:StdOut
```

Because the Shell opens standard input, standard output, and diagnostic output in the order they appear, file Found is open first, then diagnostic output is redirected to the same file. The following command has the same effect:

```
Search /NULL/ ≈.c  ≥ Found  > Dev:StdErr
```

However, if the filename and pseudo-filename specifications are simply reversed, the result is quite different:

```
Search /NULL/ ≈.c  ≥ Dev:StdOut  > Found
```

This command redirects diagnostic output to the previous standard output (probably the active window), then redirects output to file Found.

Pseudo-filenames are especially useful in a command file when you want to do something like sending standard output to the diagnostic output. Here are some examples:

```
Echo "An error message." >> Dev:StdErr
Echo "HELP !" >> Dev:Console
```

Dev:Null is useful in command files 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 command file continues to run silently. (Note that you would also need to set {Exit} to 0—see "Variables" earlier in this chapter.)

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, wiht the keyboard equivalent Command-U. When you select the menu item, 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 item 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 quotes. 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 Chapter 9.

Sample command files

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 command file 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 nice to add a disabled dotted line item to separate the new menu items from the original ones. The command file 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 command file 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}'"
            Set options "{options} '{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 4

Advanced Editing

Editing commands xx

Selections xx

Current selection x
Selection by line number x
Position x
Extending a selection x
Pattern x

Pattern matching (Using regular expressions) xx

Character expressions x
"Wildcard" operators x
Repeated Instances of regular expressions x
Tagging regular expressions with the ® operator x
Matching a pattern at the beginning or end of a line x
Inserting invisible characters x
Note on forward and backward searches x

Some useful examples xx

Transforming DumpObj output x
Finding a whole word x

This chapter describes the editing operations available as built-in commands, including the use of regular expressions. These commands enable powerful find-and-replace functions, and make it possible to automate editing operations by using command files.

Menu commands are described in Chapter 2, "Basic Editing." For a full description of the use of the command language, see Chapter 3, "Using the Command Language."

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 4-1. (They're explained in detail in Chapter 9, "Command Reference.")

Table 4-1
Editing commands

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.
Font *fontname* *fontsize* [*window*]	Change the font and/or size.
Paste [-c *count*] *selection* [*window*]	Replace *selection* with the contents of the Clipboard.
Replace [-c *count*] *selection* *replacement* [*window*]	Replace *selection* with *replacement*.
Tab *number* [*window*]	Set a window's tab value to *number* spaces
Target *name*	Make a window the target window.

If no *window* parameter is specified, editing commands act on the **target window**. The target window is the second window from the top. Therefore, to edit the active window, you'll need to switch to another window for entering your commands. (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. An implicit Find is first done to select the specified text, and 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 all regular expressions were matched. Commands run silently unless an error occurs.

Selections

Selection is a parameter to editing commands, and 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 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 4-2 shows all of the selection operators. They're fully explained in the sections following the table.

Table 4-2
Selection operators

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 fo 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
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 evaluated as explained below under "Pattern Matching."
( )	Controls order of evaluation.
Δ	Indicates position.
! and ¡	Indicates position (! = after; ¡ = before).
:	Joins two selections.

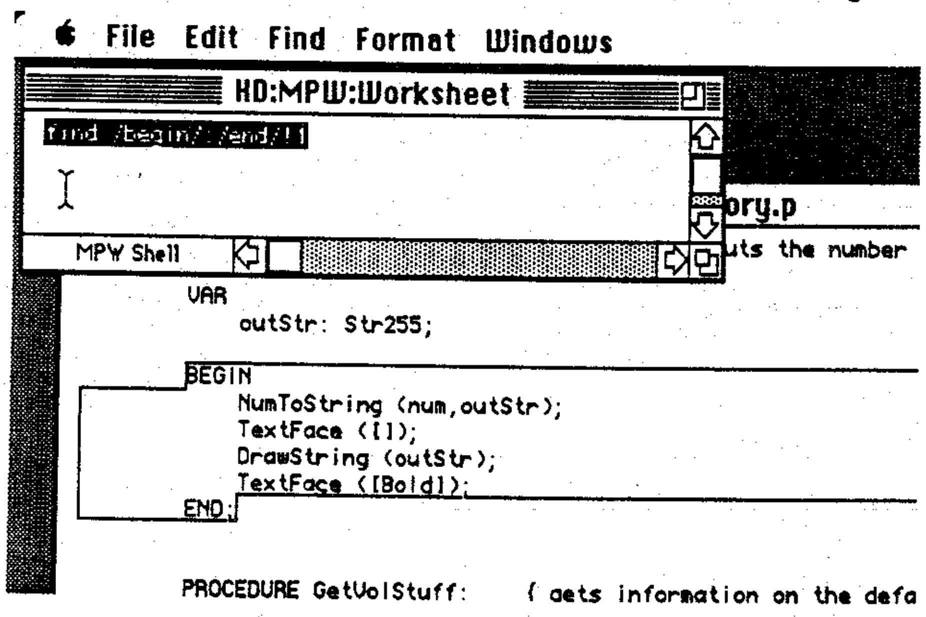
Some examples will illustrate why it's important to pay attention to the precedence of these operators:

`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 4-1



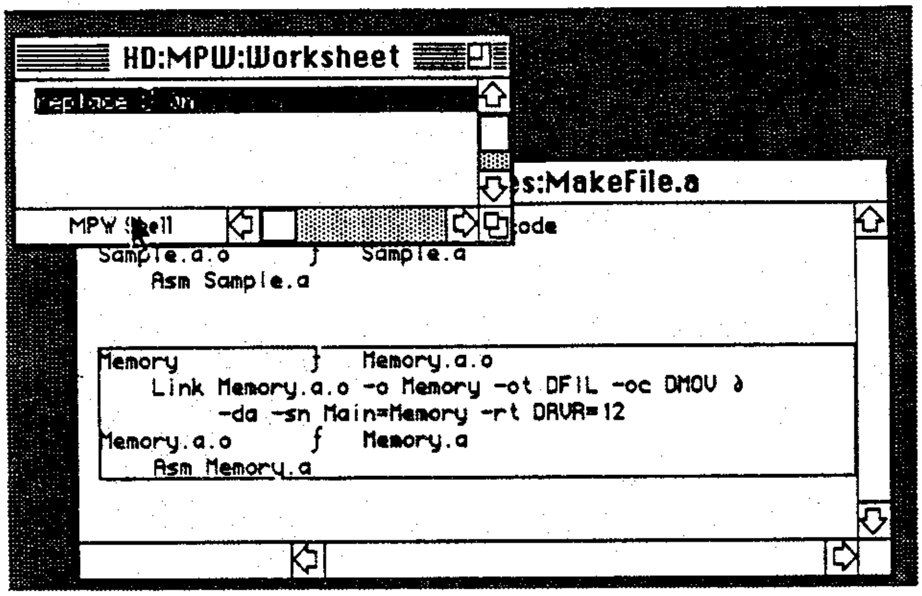
```
find /begin/:/end/!1
```

```
        VAR
            outStr: Str255;

        BEGIN
            NumToString (num,outStr);
            TextFace ([]);
            DrawString (outStr);
            TextFace ([Bold]);
        END;

        PROCEDURE GetVolStuff;    { gets information on the defa
```

Figure 4-1
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 4-2.



```
replace § ∂n
```

```
                                        ode
Sample.a.o          ƒ    Sample.a
    Asm Sample.a

Memory              ƒ    Memory.a.o
    Link Memory.a.o -o Memory -ot DFIL -oc DMOV ∂
         -da -sn Main=Memory -rt DRVR=12
Memory.a.o          ƒ    Memory.a
    Asm Memory.a
```

Figure 4-2
Selections in two windows

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 *sub*expression 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 an unquoted number as a selection, it's 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
```

The exclamation mark and inverted exclamation mark (! and ¡) specify a number of lines after or before the current selection. 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 §Δ); ¡*n* specifies a line relative to the ***start*** of the current selection (*n* lines before the line containing Δ§).

Position

A **position** is a special case of a 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.

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*.

Notice 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 file
∞ (Option-5)	Position following last character in file

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 4-1 above.) Compare this command with

```
Find /begin≈end/
```

which looks for a begin-end pair on a single line.

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 back slashes 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.)

❖ *Note:* To locate the insertion point at the beginning of the target window, for instance before executing a Find command, you can use the command

```
Find •
```

In fact, this command is so useful that you may want to add it as a menu command—see the example under the AddMenu command in Chapter 9.

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. Regular expressions are always used within the pattern delimiters /…/ or \…\.

A special set of metacharacters, called regular expression operators, are used in regular expressions (and in filename generation). The regular expression operators are listed in Table 4-3.

Table 4-3
Regular expression operators

c	Any character matches itself (unless it's one of the special characters listed below)
∂*c*	Defeat special meaning of following character (*c* is taken literally)
'…'	Literalize enclosed characters
"…"	Literalize enclosed characters, except ∂, {, and `
?	Any single character (other than return)
≈	Any string of 0 or more characters, not containing 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 beginning of line
regularExpr∞	Regular expression at end of line

These characters are considered special in the following circumstances:

∂	special everywhere except within single quotes ('…')
? ≈ * + [« ( )	special anywhere except within […], '…', and "…"
®	special only after a right parenthesis,)
•	special as first character of entire regular expression
∞	special as last character of entire regular expression
/ \	special if used to delimit a regular expression

Their precedence (from highest to lowest) is as follows:

```
( )
?  ≈  *  +  [ ]  «  ®
concatenation
•  ∞
```

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 quotes. ∂ 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 the "…" form of quotes.

"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 return

≈	any string not containing 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 3.)

A **character list** is an expression consisting of one or more characters enclosed in square brackets ([...]). It matches any character found in the list. The case-sensitivity of characters in the list is governed by the {CaseSensitive} variable (which you can set or unset by toggling the Case Sensitive menu item). 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 - characters, place them at the beginning of the list or literalize them 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 with 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 4 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 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
etc.
```

The following Replace command switches the order of the two columns:

```
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;
`[ ]+`	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 quotes are stripped off, as explained under "Quoting Special Characters" in Chapter 3. The ® operator itself can only be disabled with the escape character, ∂.

After this sequence is executed, the table would look like this:

```
456 123
456 123
456 123
456 123
```
etc.

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 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

❖ *Note:* The "Show Invisibles" menu item shows the invisible space, tab, and return characters in a file.

For more information on the escape character, see "Quoting Special Characters" in Chapter 3.

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, if the current selection is not at the end of a line, some range of characters will be selected. However, in subsequent invocations, the selection will hang at the end of the line—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 hang 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.

Some useful examples

This section shows some examples of complex use of regular expressions.

Transforming DumpObj output

The DumpObj command, described in Chapter 9, 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
```
etc.

You could position the insertion point at the beginning of the code, and use the following Replace command:

```
Replace -c ∞ /?«41»/ "∂t∂t" # delete everything up to the instruction
```

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 (Δ*selection*!1), and using 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}/
```

Chapter 5

Editing Resources With ResEdit

About ResEdit xx

Uses x
Extensibility x

Using ResEdit xx

Working with files x
Working within a file x
Working within a resource type x
Editing individual resources x
    'CURS' resources x
    'DITL' resources x
    'FONT' resources x
    'ICN#' resources x

Creating a resource template xx

The chapter describes ResEdit, a stand-alone application for editing resources.

❖ *Note:* As in ***Inside Macintosh,*** resource types are shown within single quotes; for example, 'STR ' (that is, STR*space*). The quotes are not part of the name.

About ResEdit

ResEdit is an interactive, graphically based application for manipulating the various resources in a Macintosh application. It lets you create and edit all standard resource types except 'CODE', and copy and paste all resource types (including 'CODE'). ResEdit actually includes a number of different resource editors: there is a general resource editor, for editing any resource in hex and ASCII format, and there are several individual resource editors for specific types of resources. You can also create your own resource editors to use with ResEdit.

Uses

ResEdit is especially useful for creating and changing graphic resources such as dialogs and icons. For example, you can use ResEdit to put together a quick prototype of a user interface and try out different formats and presentations of resources. You can also use ResEdit for translating resources into a foreign language without having to recompile the program. You can use ResEdit to modify a program's resources at any stage in the process of program development.

Once you have created or modified a resource with ResEdit, you can use the Resource Decompiler, DeRez, to convert the resource to a textual representation that can be processed by the Resource Compiler, Rez. You can then add comments to this text file or otherwise modify it with the Shell editor. (Rez and DeRez are fully described in the next chapter.)

Extensibility

A key feature of ResEdit is its extensibility. Because it can't anticipate the format of all the different types of resources that you might use, ResEdit has been designed so that you can teach it to recognize and parse new resource types.

There are two ways that you can extend ResEdit to handle new types:

- You can create templates for your own resource types. ResEdit lets you edit most resource types by filling in the fields of a dialog box—this is the way you edit 'BNDL' and 'FREF' resources, for example. The layout of these dialog boxes is determined from a template in ResEdit's resource file, and you can add templates to edit new resource types. Resource templates are described later in this chapter.
- You can also program your own special-purpose resource picker and/or editor and then add it to ResEdit. The **resource picker** is the code that displays all the resources of one type in the resource type window. The **editor** is the code that displays and allows you to edit a particular resource. These pieces of code are separate from the main code of ResEdit. A set of Pascal routines, called ResEd, is available for this purpose—see *MPW Reference* for information.

Using ResEdit

From the MPW Shell, you can start ResEdit by entering the command

```
ResEdit
```

(This assumes, of course, that ResEdit is in the Applications folder, or elsewhere in the search path defined by the {Commands} variable.) From the Finder, you can just select and open the ResEdit icon. ResEdit displays a window that lists the files and folders for each disk volume currently mounted (Figure 5-1).

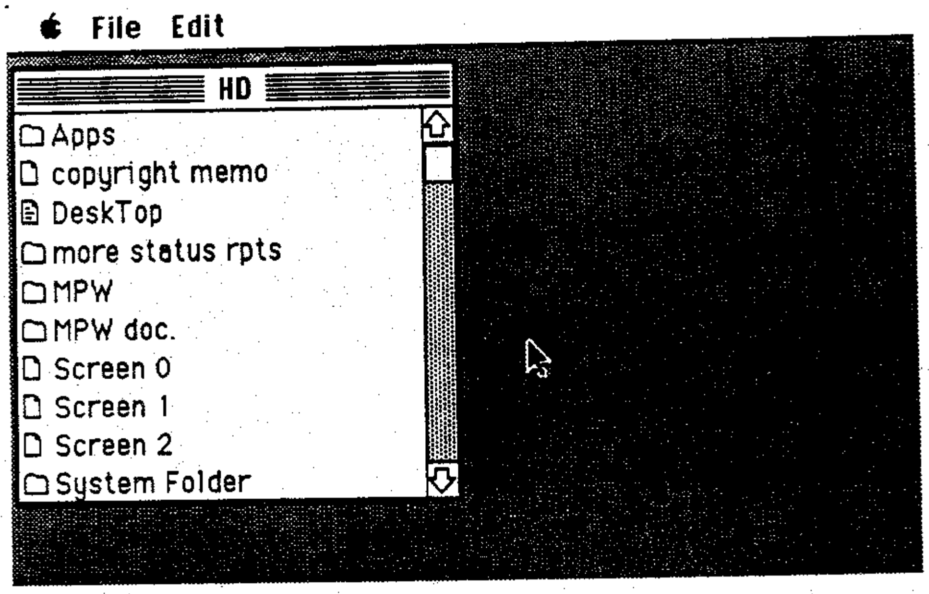


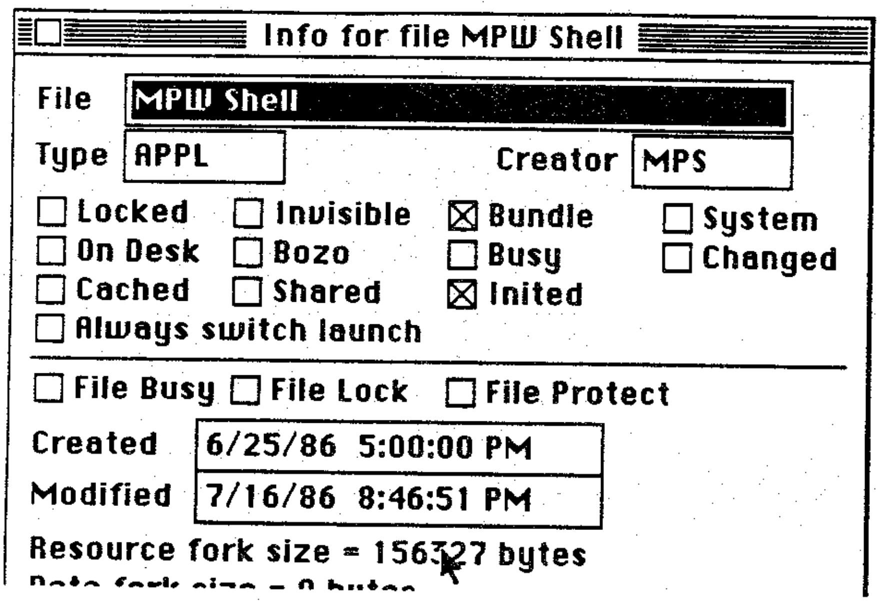
Figure 5-1
A disk volume window

Working with files

To list the resource types in a file, select and open the filename from the list. (You can select a name by clicking on it or by typing one or more characters of the name.)

When a directory window is the active window, the File menu commands act as follows:

New	Creates a new file.
Open	Opens the selected file or folder (this is the same as double-clicking on the name).
Close	Closes the volume window (this is the same as clicking the close box). If it's a 3.5-inch disk, the disk is ejected.
Get Info	Displays file information and allows you to change it. For example:



Info for file MPW Shell

File	MPW Shell		
Type	APPL	Creator	MPS

☐ Locked ☐ Invisible ☒ Bundle ☐ System
☐ On Desk ☐ Bozo ☐ Busy ☐ Changed
☐ Cached ☐ Shared ☒ Inited
☐ Always switch launch

☐ File Busy ☐ File Lock ☐ File Protect

Created	6/25/86 5:00:00 PM
Modified	7/16/86 8:46:51 PM

Resource fork size = 156327 bytes

Transfer	Allows you to transfer to an application other than the application that launched ResEdit. (This is an alternative to the Quit command.)
Quit	Quits from ResEdit and returns to the MPW Shell (or Finder).

Warning

You can edit any file shown in the window, including the System file and ResEdit itself. However, it's dangerous to edit a file that's currently in use. Edit a copy of the file instead. (For example, edit the System file on a non-boot volume.)

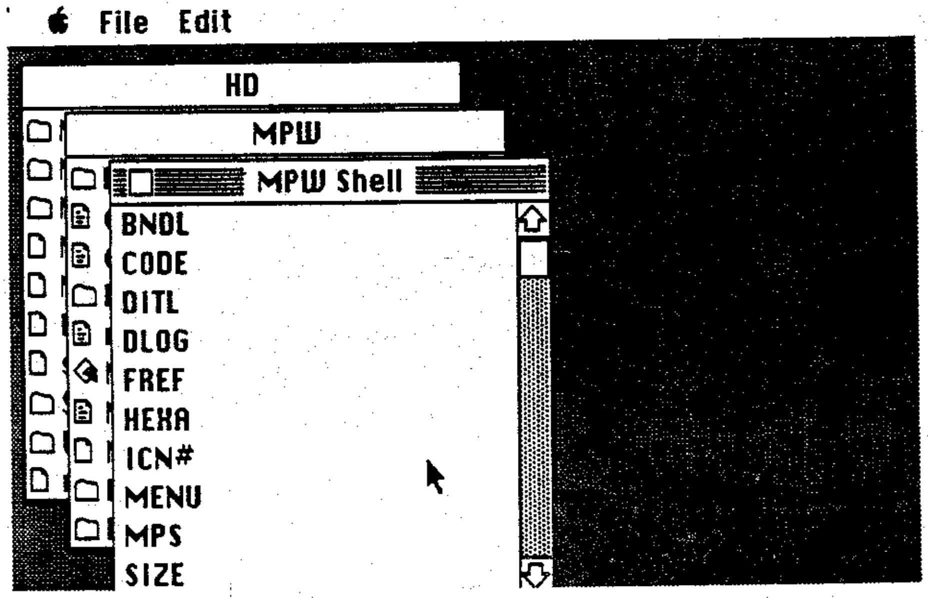
ResEdit will recognize a new disk when it's inserted, and also handles multiple drives. Note that you can also use ResEdit to copy or delete files:

- To delete a file, select the file and choose Clear from the Edit menu.
- To copy a resource file, you must select all of its resources and copy them. Then paste them into a new file. (File attributes are not automatically copied by this operation—you must set them via the Get Info command.) ResEdit cannot copy a data fork.

Working within a file

When you open a file, a window displays a list of all the resource types in that file (Figure 5-2). While this window is the active window, you can create new resources, copy or delete existing resources, and paste resources from other files.

❖ *Note:* The resources are displayed by a resource picker. The general resource picker displays the resources by type, name, and ID number; there are also special resource pickers for some resource types (for example, the 'ICON' resource picker displays the icons graphically).



BNDL
CODE
DITL
DLOG
FREF
HEXA
ICN#
MENU
MPS
SIZE

Figure 5-2
A file window

When a file window is the active window, the File menu commands have the following effects:

New	Creates a new resource in the open file.
Open	Opens a window displaying all resources of the resource type selected. (Select the resource type by clicking on it or by typing its first character.)

	Note: If you hold down the Option key while opening a resource type, the resource window will open with the general resource picker.
Open General	Opens the general resource picker.
Close	Closes the file window and asks if you want to save the changes you have made.
	Note: If you've made changes, you should not reboot before closing.
Revert	Changes the resource file back to the version that was last saved to disk.
Quit	Quits from ResEdit.

When a file window is the active window, the Edit menu commands have the following effects:

Cut	Removes all resources of the resource types selected, placing them in the ResEdit scrap.
Copy	Copies all resources of the resource types selected into the ResEdit scrap.
Paste	Copies the resources from the ResEdit scrap into the file window's resource type list.
Clear	Removes all resources of the resource type selected, without placing them in the ResEdit scrap.
Duplicate	Creates duplicates of all resources of the resource types selected, and assigns a unique resource ID number to each new resource.

Working within a resource type

Opening a resource type produces a window that lists each resource of that type in the file (Figure 5-3). This list will take different forms, depending on the particular resource picker; if you hold down the Option key during the open, the general resource picker is invoked.

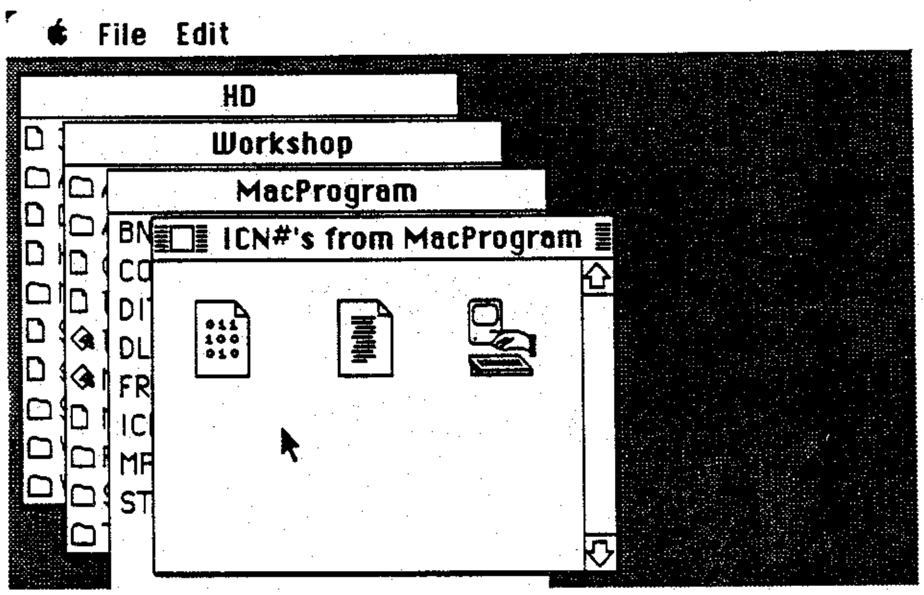
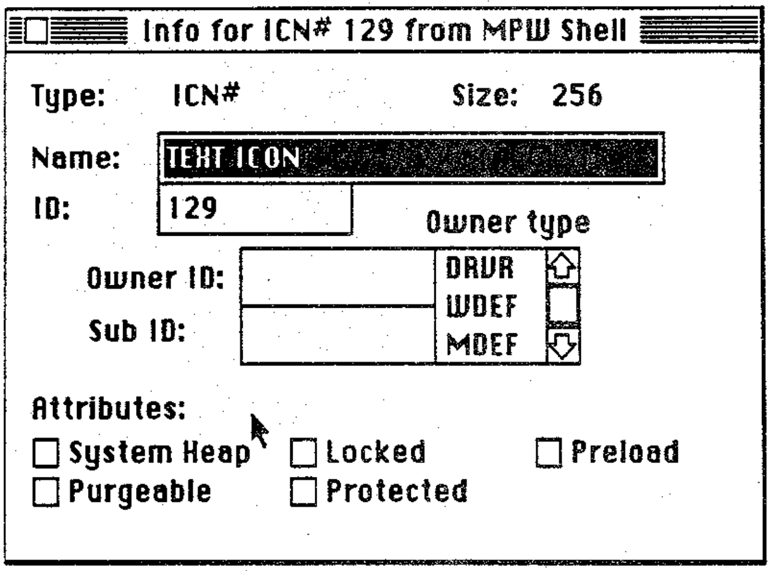


Figure 5-3
A resource type window

When a resource type window is the active window, the File menu commands have the following effects:

New	Creates a new resource and opens its editor.
Open	Opens the appropriate editor for the resource you selected.
Open as...	Allows you to open an editor template of a different type.
Open General	Opens the general (hex) resource editor.
Close	Closes the resource type window.
Revert	Changes the file back to what it was before opening the resource type window.

Get Info	Displays resource information and allows you to change it. For example:


Info for ICN# 129 from MPW Shell

Type: ICN#   Size: 256

Name: TEXT ICON

ID: 129

Owner type: DRVR, WDEF, MDEF

Owner ID:

Sub ID:

Attributes:
☐ System Heap ☐ Locked ☐ Preload
☐ Purgeable ☐ Protected

When a resource type window is the active window, the Edit menu commands have the following effects:

Undo	Undoes the most recent editing command. Undo may or may not be selectable, depending on the specific editor in use.
Cut	Removes the resources that are selected, placing them in the ResEdit scrap.
Copy	Copies all the resources that are selected into the ResEdit scrap.
Paste	Copies the resources from the ResEdit scrap into the resource type window.
Clear	Removes the resources that are selected, without placing them in the ResEdit scrap.
Duplicate	Creates a duplicate of the selected resources and assigns a unique resource ID number to each new resource.

Editing individual resources

To open an editor for a particular resource, either double-click on the resource or select it and choose Open from the File menu. One or more auxiliary menus may appear, depending on the type of resource you're editing. Some editors, such as the 'DITL' editor, allow you to open additional editors for the elements within the resource. All the editors use File and Edit menus similar to those described above, but operate on individual resources or individual elements of a resource.

If you hold down the Option key while opening a resource, the **general data editor** is invoked. This editor allows you to edit the resource as hexadecimal or ASCII data. If you hold down the Shift and the Option keys while opening, ResEdit shows you a list of all editors and templates.

Caution

Individual editors may not be appropriate for all resource types—inappropriate editors may cause system errors to occur.

The menus for some of the editors are discussed below. The use of the editors not discussed here should be apparent when you run them.

❖ *Note:* The general data editor will not edit resources larger than 16K bytes; however, you can *move* larger resources with the Cut, Copy, Paste, and Clear commands as described above.

'CURS' (Cursor) resources

For 'CURS' resources, the editor displays three images of the cursor (Figure 5-4). You can manipulate all three images with the mouse.

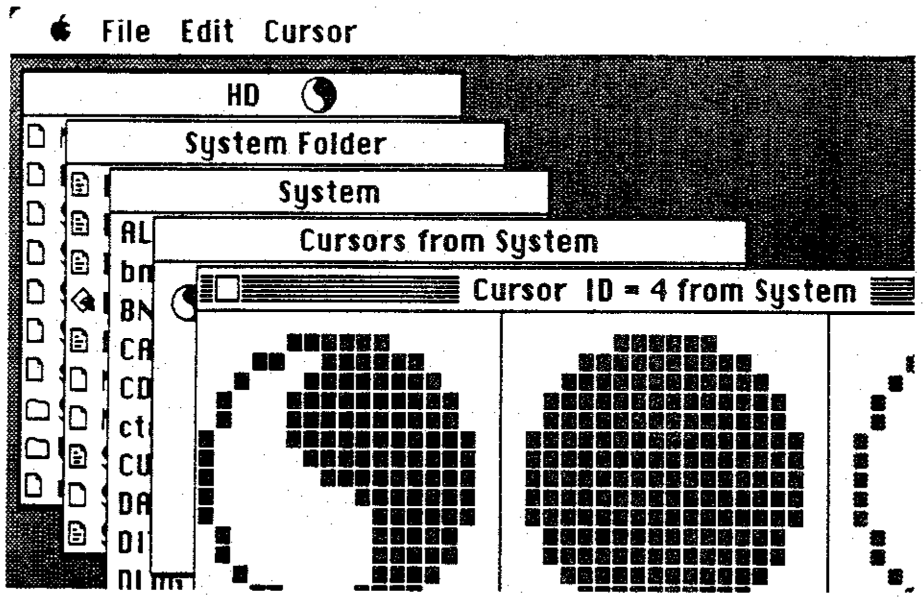


Figure 5-4
Editing 'CURS' resources

The left image shows how the cursor will appear. The middle image is the mask for the cursor, which affects how the cursor appears on various backgrounds. The right image shows a gray picture of the cursor with a single point in black—this point is the cursor's hot spot.

The Cursor menu contains the following commands:

Try Cursor	Lets you try out the cursor by having it become the cursor in use. (Restore Arrow restores the standard arrow cursor.)
Data ->Mask	Copies the cursor image to the mask editing area.

'DITL' (Dialog Item List) resources

For 'DITL' resources, the editor displays an image of the item list as your program would display it in a dialog or alert box. When you select an item, a size box appears in the lower-right corner of its enclosing rectangle so that you can change the size of the rectangle. You can move an item by dragging it with the mouse.

If you open an item within the dialog box, the editor associated with the item is invoked; for an 'ICON', for example, the icon editor is invoked. If you hold down the Option key while opening, the general data editor is invoked.

The DITL menu contains the following commands:

Bring to Front	Allows you to change the order of items in the item list. Bring to Front causes the selected item to become the last (highest numbered) item in the list. The actual number of the item is shown by the 'DITM' editor.
Send to Back	Like Bring to Front, except that it makes the selected item the first item in the list—that is, item number 1.
Grid	Aligns the item on an invisible 8-pixel by 8-pixel grid. If you change the item location while Grid is on, the location will be adjusted such that the upper-left corner lies on the nearest grid point above and to the left of the location you gave it. If you change the size, it will be made a multiple of 8 pixels in both dimensions.
Use RSRC Rect	Restores the enclosing rectangle to the rectangle size stored in the underlying resource. Note that this works on 'ICON', 'PICT', and 'CNTL' items only; the other items have no underlying resources.
Use full window	Adjusts the window size so that all items in the item list are visible in the window.

'FONT' resources

For 'FONT' resources, the editor window is divided into three panels: a sample text panel, a character selection panel, and a character editing panel. These are shown in Figure 5-5.

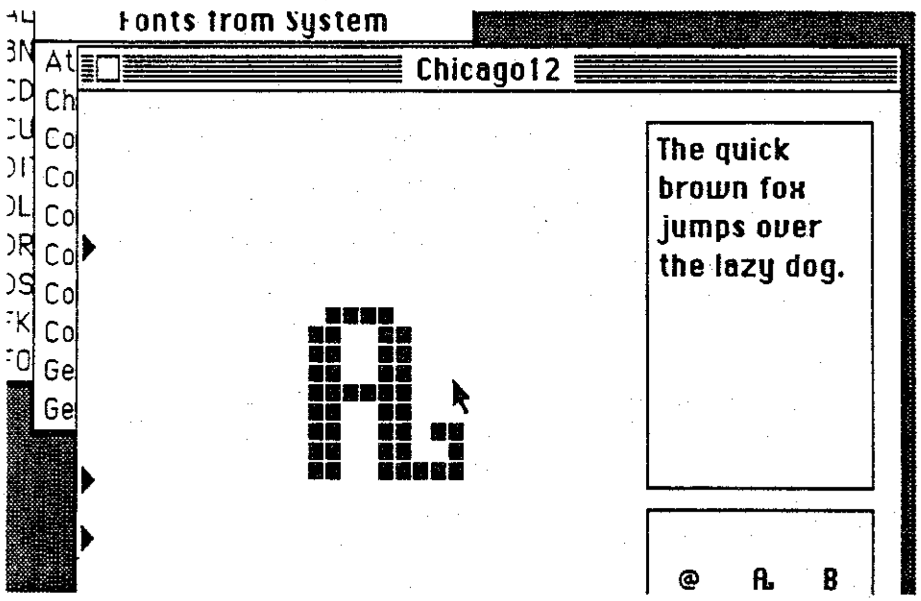


Figure 5-5
FONT editor window

The **sample text panel**, at the upper right, displays a sample of text in the font being edited. (You can change this text by clicking in the text panel and using normal Macintosh editing techniques.)

The **character selection panel** is below the text panel. You can select a character to edit by typing it (using the Shift and Option keys if necessary), or by clicking on it in the row of three characters shown. (Click on the right character in the row to move upward through the ASCII range; click on the left character to move downward.) The character you select is boxed in the center of the row with its ASCII value shown below it (in decimal).

The **character editing panel** on the left side of the window shows an enlargement of the selected character. You edit it, like FatBits in MacPaint, by clicking bits on and off. The black triangles at the bottom of the character editing panel set the left and right bounds (that is, the character width). The three triangles at the left of the panel control the ascent, baseline, and descent.

Caution

Changing the ascent or descent of a character changes the ascent or descent for the entire font.

Any changes you make in the character editing panel are reflected in the text panel and the character selection panel. Remember that you cannot save the changes until you close the file.

You can also change the name of a font. The font name is stored as the name of the resource of that font family with size 0. This resource does not show up in the normal display of all fonts in a file. To display it, hold down the Option key while you open the FONT type from the file window. You will see a generic list of fonts. Select the font with the name you wish to change and choose Get Info.

'ICN#' (Icon List) resources

For 'ICN#' resources, the editor displays two panels in the window (Figure 5-6). The upper panel is used to edit the icon. It contains an enlargement of the icon on the left and an enlargement of the icon's mask on the right. The lower panel shows, from left to right, how the icon will look unselected, selected, and open on both a white and a gray background. It also shows how the icon will appear in the Finder's small icon view.

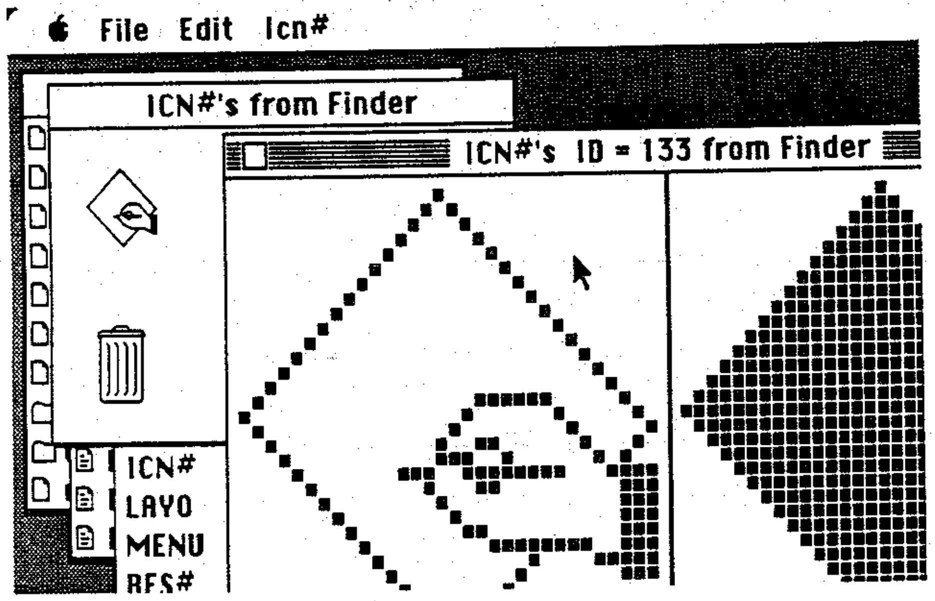


Figure 5-6
'ICN#' Window

To install a new icon for your application when you already have an old one in the Finder's desktop file, follow these steps:

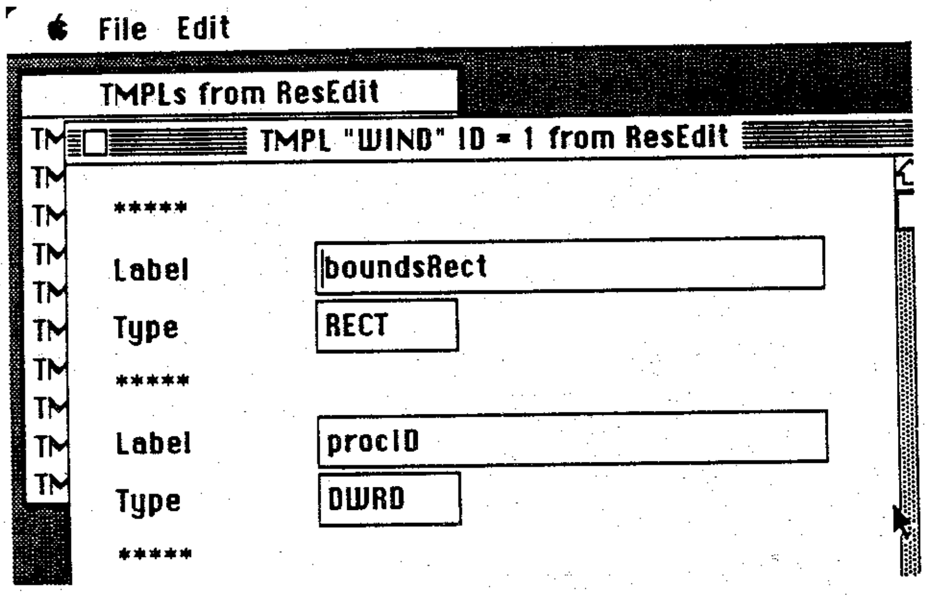
1. Open the file called DeskTop.
2. Open type 'BNDL' and find the bundle that belongs to your application. (This is the one that has your owner name in it.) Look through the bundle and mark down the type and resource ID of all resources bundled together by the bundle (that is, the 'ICN#'s and 'FREF's).
3. Go back to the DeskTop window and remove these resources along with your 'BNDL' and signature resource (the resource whose type is your application's signature).
4. Now close the DeskTop window, save changes, and quit ResEdit. Your new icon will be installed if you have the proper 'BNDL', 'FREF', and 'ICN#' resource numberings.

   Note: To see how 'BNDL', 'FREF', and 'ICN#' resources are interrelated, use ResEdit to look at those resources in an existing application such as the MPW Shell.

Alternatively, you can rebuild the DeskTop file by holding down the Option and Command keys when entering the Finder. (This method is faster and easier, but you will lose your Finder Get Info comments; you will also lose folder names on a non-HFS volume.)

Creating a resource template

You can customize ResEdit by creating new templates for your own resource types. The generic way of editing a resource is to fill in the fields of a dialog box—for example, this is the way you edit 'FREF', 'BNDL', and 'STR#' resources. The layout of these dialog boxes is set by a template in ResEdit's resource file. The template specifies the format of the resource and also specifies what labels should be put beside the editText items in the dialog box that's used for editing the resource. You can find these templates by opening the ResEdit file and then opening the type window for 'TMPL' resources. For example, if you open the template for 'WIND' resources (this is the 'TMPL' with name "WIND"), you'll see the template shown in Figure 5-7.



```
*****
Label     boundsRect
Type      RECT
*****
Label     procID
Type      DWRD
*****
```

Figure 5-7
Window template data

The window template, then, consists of the following:

1. A RECT (4 words) specifying the boundary of the window.
2. A word that is the procID for the window (DWRD tells ResEdit to display the word in decimal as opposed to hex).
3. A Boolean indicating whether or not the window is visible (BOOL is 2 bytes in the resource but is displayed as a radio button in the dialog window used for editing).
4. Another Boolean indicating whether or not the window has a close box.
5. A long word that is the reference value (refCon) for the window (DLNG indicates that it should be displayed in the editor as a decimal number).
6. A Pascal string (PSTR), the title of the window.

You can look through the other templates and compare them with the structure of those resources to get a feel for how you might define your own resource template. (These templates are equivalent to the resource type declarations contained in the {RIncludes} directory—refer also to the DeRez command in Chapter 9.)

These are the types you have to choose from for your editable data fields:

DBYT, DWRD, DLNG	decimal byte, word, long word

HBYT, HWRD, HLNG	hex byte, word, long word
HEXD	hex dump of remaining bytes in resource
PSTR	a Pascal string (length byte followed by the characters)
LSTR	long string (length long followed by the characters)
WSTR	same as LSTR, but a word rather than a long word
ESTR, OSTR	Pascal string padded to even or odd length (needed for DITL resources)
CSTR	a C string
ECST, OCST	even-padded C string, or odd-padded C string (padded with nulls)
BOOL	Boolean
BBIT	binary bit
TNAM	type name (4 characters, like OSType and ResType)
CHAR	a single character
RECT	an eight-byte rectangle
H*nnn*	3-digit hex number (where *nnn* < $900); displays *nnn* bytes in hex format.

ResEdit will do the appropriate type checking for you when you put the editing dialog window away.

The template mechanism is flexible enough to describe a repeating sequence of items within a resource, as in 'STR#', 'DITL', and 'MENU' resources. You can also have repeating sequences within repeating sequences, as in 'BNDL' resources. To terminate a repeating sequence, put the appropriate code in the template as follows:

LSTZ ... LSTE	*List Zero–List End.* Terminated by a 0 byte (as in 'MENU's).
ZCNT LSTC ... LSTE	*Zero Count/List Count–List End.* Terminated by a zero-based count that starts the sequence (as in 'DITL' resources).
OCNT LSTC ... LSTE	*One Count/List Count–List End.* Terminated by a one-based count that starts the sequence (as in 'STR#' resources).

LSTB	
...	
LSTE	Ends at the end of the resource (no example exists in the given templates).

The "list-begin" code begins the repeating sequence of items, and the LSTE code is the end. Labels for these codes are usually set to the string "******". Both of these codes are required.

To create your own template, follow these steps:

1. Open the ResEdit file window.
2. Open the 'TMPL' type window.
3. Choose New from the File menu.
4. Select the list separator ("****").
5. Choose New from the File menu. You may now begin entering the *label,type* pairs that define the template. Before closing the template editing window, choose Get Info from the File menu and set the name of the template to the four-character name of your resource type.
6. Close the ResEdit file window and save changes.

The next time you try to edit or create a resource of this new type, you'll get the dialog box in the format you have specified.

Warning

Changing resource templates (and hence resource type descriptions) can cause system crashes if you open older versions of a resource with a new template.

Chapter 6

Resource Compiler and Decompiler

About the Resource Compiler and Decompiler xx

Resource Decompiler x
Standard type declaration files x
Using Rez and DeRez x

Structure of a resource description file xx

Sample resource description file x

Resource Description Statements xx

Syntax notation x
    Special terms x
Include — include resources from another file x
    Syntax x
    Resource attributes x
Read — read data as a resource x
    Syntax x
    Description x
Data — specify raw data x
    Syntax x
    Description x
Type — declare resource type x
    Syntax x
    Description x
    Data-type specifications x
        Numeric types x
        Boolean type x

        Character type x
        String type x
        Point and rectangle types x
    Fill and align types x
        Fill specification x
        Align specification x
    Array type x
    Switch type x
    Sample type statement x
        An aside: symbol definitions x
Resource — specify resource data x
    Syntax x
    Description x
    Data statements x
        Switch data x
        Array data x
    Sample resource definition x
        Symbolic names x

Preprocessor directives xx

Variable definitions x
Include directives x
If-Then-Else processing x

Resource description syntax xx

Numbers and literals x
Expressions x
Variables x
Strings x
    Escape characters x

In the Macintosh Programmer's Workshop, you can build a resource graphically with ResEdit, or in text form with the Resource Compiler. 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 Chapter 9. General information on resource files is given in the "Resource Manager" chapter of ***Inside Macintosh***.

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 6-1 illustrates the complementary relationship between Rez and DeRez.

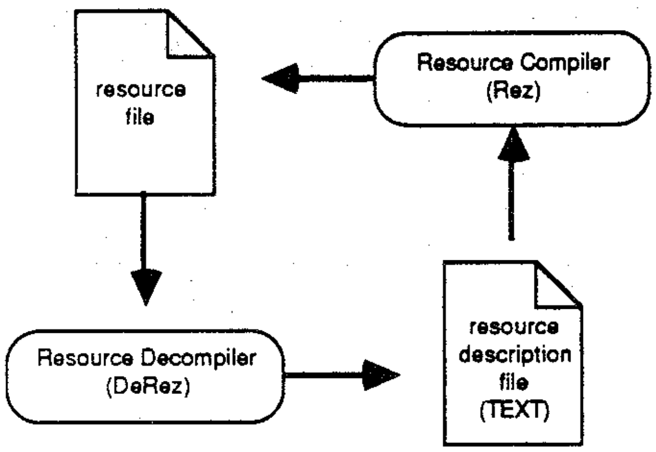


Figure 6-1
Rez and DeRez

The Resource Compiler can combine resources or resource descriptions from a number of files into a single resource file. The Resource Compiler also supports preprocessor directives that allow you to substitute macros, include other files, and use if-then-else constructs. (See "Preprocessor Directives" later in this chapter.)

Resource Decompiler

The DeRez command 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 is in 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 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 Pascal also includes a sample tool, ResEqual, which directly compares resource files.

Standard type declaration files

Three text files, Types.r, SysTypes.r, and MPWTypes.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', and 'NFMT'
MPWTypes.r	Type declarations for the MPW driver type 'DRVW'

Using Rez and DeRez

Rez and DeRez are primarily used to create and modify resource files. Figure 6-2 illustrates the process of creating a resource file.

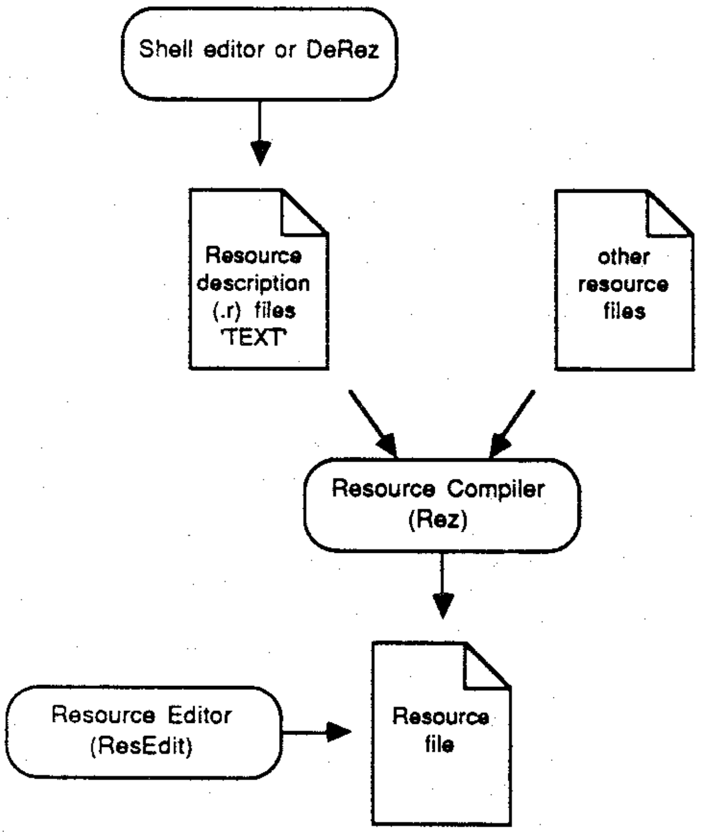


Figure 6-2
Creating a resource file

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'd use Rez to combine the Linker's output with other resources to create an executable program file. (See Chapter 7 for details.)

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 has no built-in resource types—you need to define your own types, or include the appropriate ".r" files.

A resource description file may contain of any number of statements, where a statement is any of the following:

`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.

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 anywhere where white space is allowed in a resource description file, within the comment delimiters /* and */.
- **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 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 the type declaration in {RIncludes}Types.r.

```
DeRez Sample -only WIND  Types.r  > DeRez.Out
```

After executing this command, DeRez.Out would contain the following:

```
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 following reference section.

Resource description statements

This section describes the syntax and use of the five types of resource description statements: `include`, `read`, `data`, `type`, and `resource`.

Syntax notation

The syntax notation in this chapter follows the conventions given in the Preface. Additionally, the following conventions are used:

- Words that are part of the resource description language are shown in `courier` to distinguish them from other text. The Resource Compiler 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 quotes ('...'); 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

`Include` statements can have the following forms:

- `include` *file* [*resource-type* ['(' *resource-name* | *ID* ')']] ;

  Read the resource of type *resource-type* with the specified resource name or resource ID 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* `')'`
  `as` *resource-type2* `'('` *ID* [, *resource-name*] [, *attributes*...] `')'` `;`

  Read the resource of type *resource-type1* with the specified name or ID 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 */
```

Resource attributes

You can specify *attributes* as a numeric expression (see the "Resource Manager" chapter 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 list 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

`Data` statements let you 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
declare
        char
        string
        pstring
        cstring
        point
        rect
```

***data-types*:** Data-type statements

a field of the given data type. They can also associate symbolic names or constant values with the datatype.

`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 following sections.

❖ *Note:* Several of these types require additional fields—the exact syntax is given in the following sections.

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* ;

Data-type specifications

Data-type specifications can take three forms, as shown in the following 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 by 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.

❖ *Note:* 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 as follows:

[`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 may 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`	65535	−32768
`longint`	4294967295	−2147483648

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*...];

Type `boolean` declares a 1-bit field; this is equivalent to

```
unsigned bitstring[1]
```

Note that this type is not the same as a boolean variable as defined by Pascal.

Character type. Characters are declared as follows:

`char` [*symbolic-value*...];

Type `char` declares an 8-bit field (this is the same as writing `string[1]`).

An example follows:

```
type 'SYMB'  {
    char  dollar = "$",percent = "%";
};
resource 'SYMB' (128)  {
    dollar
};
```

String type. String data types are specified as follows:

string-type ['[' *length* ']'] [*symbolic-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. Type `char` is shorthand for `string[1]`.
`pstring`	Pascal string (a leading byte containing the length information is generated), aligned to a word boundary. `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.

`cstring`	C string (a trailing null byte is generated). `Cstring[`*n*`]` contains *n*–1 characters and is *n* bytes long. A `cstring` of length 1 can be assigned only the value `""`, because `cstring [1]` has room only for the terminating null.

Each 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 $1 \le$ ***length*** ≤ 2147483647 for `string` and `cstring`, and in the range $1\le$*length*≤ 255 for `pstring`.

❖ *Note:* You cannot assign the value of a literal to a string-type.

If no length indicator is given, a `pstring` 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.

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 top-left and bottom-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. Fill and alignment 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 as follows:

`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 the following 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 must refer to that array in a constant expression with the `$$countof(`***array-name***`)` function; otherwise DeRez will be unable to decompile resources of this type. 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.

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is as follows:

`switch '{' `*case-statement...*` '}';`

where a *case-statement* has the following form:

`case `*case-name*` : [ `*case-body*` ; ]...`

Case-name is a string. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in the following 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;
        ...et cetera...
    };
};
```

Sample type statement

The following sample type statement is the standard declaration for a 'WIND' resource, taken from the Types.r file:

```
type 'WIND'{
                                                      /* boundsRect */
        rect;
        integer       documentProc, dBoxProc, plainDBox, /* procID */
                      altDBoxProc, noGrowDocProc,
                      zoomProc=8, rDocProc=16;
        byte          invisible, visible;             /* visible */
        fill byte;
        byte          noGoAway, goAway;               /* has 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 "Resource Manager" chapter 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.

An aside: 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;
```

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, char`
`bitstring`	`boolean, byte, integer, longint, bitstring`
`rect`	`rect`
`point`	`point`

Switch data. Switch data statements are specified by using the following 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 the following 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 "Window Manager" chapter of ***Inside Macintosh*** for information about resources in windows.)

Here, again, is the type declaration given above under "Sample Type Statement":

```
type 'WIND'{
        rect;                                                   /* boundsRect */
        integer      documentProc, dBoxProc, plainDBox,         /* procID */
                     altDBoxProc, noGrowDocProc,
                     zoomProc=8, rDocProc=16;
        byte         invisible, visible;                        /* visible */
        fill byte;
        byte         noGoAway, goAway;                          /* has 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 be 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 as follows:

    {*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 zero, 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,
```

Preprocessor directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other Rez processing takes place. This section describes the preprocessor directives.

The syntax of the preprocessor is very similar to the C-language preprocessor. Each of the preprocessor statements must be expressed on a single line, beginning on a new line and terminated by a Return character. ***Identifiers*** (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_). Identifiers may not start with a digit. Identifiers are not case sensitive. An identifier may be any length.

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* is to be replaced with the text *data*. A macro can be extended 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

Include directives

The #include directive reads a text file:

#include *file*

Include the text file *file*. The maximum nesting is to 10 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

The following directives provide conditional processing:

#if *expression*
[#elif *expression*]
#endif

❖ *Note:* *expression* is defined later in this chapter; with the #if and #elif directives, *expression* may also include the following expression:

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 (Japanese)
    "Konnichiwa"
#endif
};
```

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

Decimal	*nnn*...	Signed decimal constant between 4294967295 and −2147483648.
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 0B11111111111111111111111111111111 and 0B10000000000000000000000000000000.

Literal	*'aaaa'*	A literal can contain one to four characters. Characters are printable ASCII characters or escape characters (defined below). 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 6-3

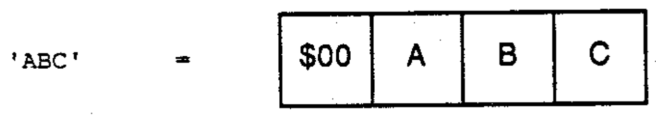


Figure 6-3
Padding of Literals

Expressions

An expression can consist of simply a number or literal. Expressions can also include numeric variables and the system function:

`$$countof` '(' *array-name* ')'

Expressions can include the expression operators listed in Table 6-1. Table 6-1 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 6-1
Resource Description File 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.
	expr1 < *expr2*	Less than.
	expr1 <= *expr2*	Less than or equal.
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 !, >, >=, <, <=, ==, !=, &&, | | evaluate to 1 (true) or 0 (false).

Variables

There are some Resource Compiler variables that contain commonly used values. All Resource Compiler variables start with $$ followed by an alphanumeric identifier.

The following variables have string values (typical values are given in parentheses):

`$$Version`	Version number of the Resource Compiler. ("V1.0 ")
`$$Date`	Current date. Useful for putting timestamps into the resource file. The format is generated through the ROM call IUDateString. ("Thursday, February 20, 1986")
`$$Time`	Current time. Useful for timestamping the resource file. The format is generated through the ROM call IUTimeString. ("7:50:54 AM")
`$$Shell("`*strExpr*`")`	Current value of the exported Shell variable {*strExpr*}. Note that the curly braces must be omitted, and the double quotes must be present.
`$$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.

The following variables have numeric values:

`$$Hour`	Current hour. Range 0–23.
`$$Minute`	Current minute. Range 0–59.
`$$Second`	Current second. Range 0–59.
`$$Year`	Current year.
`$$Month`	Current month. Range 1–12.
`$$Day`	Current day. Range 1–31.
`$$Weekday`	Current day of the week. Range 1–7 (that is, Sunday – Saturday).

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 6-3.) 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 as white space.)

Figure 6-4 shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello";
```

$05	H	e	l	l	o	$00	$00	$00	$00	$00

Figure 6-4
Internal representation of a Pascal string

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line; the semicolon (`;`) terminates 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 in a string, precede the quotation mark with a backslash (\").

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 Return character in a string, use the escape sequence

```
\r
```

Valid escape sequences are given in Table 6-3.

Table 6-2
Resource Compiler escape sequences

Escape sequence	Name	Hex value	Printable equivalent
`\t`	tab	$09	none
`\b`	backspace	$08	none
`\r`	return	$0D	none
`\n`	newline	$0D	none
`\f`	form feed	$0C	none
`\v`	vertical tab	$0B	none
`\?`	rubout	$7F	none
`\\`	backslash	$5C	\
`\'`	single quote	$3A	'
`\"`	double quote	$22	"

❖ *Note:* On the Macintosh, newline is identical to return.

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are as follows:

Base	Form	Number of 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`

Some examples are

```
\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.

Chapter 7

Putting Together an Application, MPW Tool, or Desk Accessory

Contents

Overview of the Build Process xx

The Structure of a Macintosh Application xx

Linking xx

What to link with x
Linking multilingual programs x

File Types and Creators xx

Putting Together an MPW Tool xx

Putting Together a Desk Accessory or Driver xx

Linking a desk accessory or driver x
The desk accessory resource file x

Using Make xx

Format of a makeFile x
Dependency rules x
- Double-*f* dependency rules x
- Default rules x
  - Built-in default rules x
  - Directory dependency rules x

Variables in makefiles x
- Shell variables x
- Defining variables within a makefile x
- Built-in Make variables x

Quoting in makefiles x
Comments in makefiles x
Executing Make's output x
Debugging makefiles x
An example x

More About Linking xx

Linker functions x
Segmentation x
    Segments with special treatment x
Setting resource attributes x
Controlling the numbering of code resources x
Resolving symbol definitions x
    Multiple external symbol definitions x
    Unresolved external symbols x
Linker location map x
Optimizing your links x

Library Construction xx

Using Lib to build a specialized library x
    Removing unreferenced modules x
    Guidelines for choosing files for a specialized library x

Overview of the build process

This chapter describes the mechanics of building a program—the steps involved are nearly the same for applications, MPW tools, desk accessories, and drivers. All programmers should read the opening sections of this chapter, which explain the entire build process for an application, the standard case. Later sections explain what's different about building an MPW tool, desk accessory, or driver.

Building a program consists of the following steps:

1. **Create source files and compile them.** Each source file is compiled or assembled to produce a corresponding object file. (For information on writing programs in Pascal, C, or assembly language, and including the proper interface or include files, see the appropriate MPW language manual.)
2. **Create additional 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). See Chapters 5 and 6 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).

   Note: For building a desk accessory or driver in Pascal or C, an additional step is required—run Rez to create the final 'DRVR' resource. For details, see "Putting Together a Desk Accessory or Driver," later in this chapter.

Figure 7-1 illustrates the complete process.

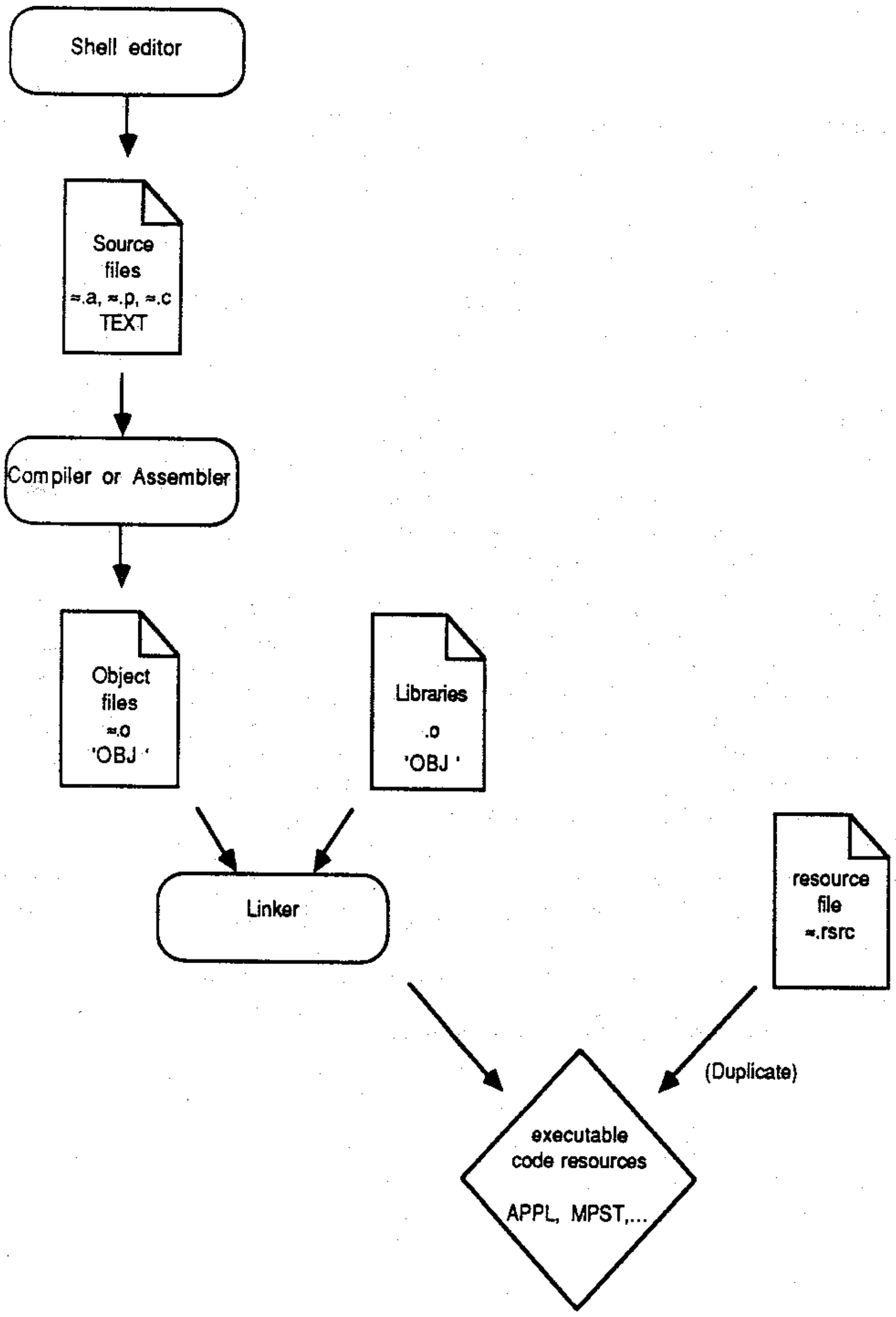

Shell editor
Source files
≈.a, ≈.p, ≈.c
TEXT
Compiler or Assembler
Object files
≈.o
'OBJ '
Libraries
.o
'OBJ '
Linker
resource file
≈.rsrc
(Duplicate)
executable code resources
APPL, MPST,...

Figure 7-1
Building a program

For example, the following series of commands compile, "Rez," 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 the "Using Make" section later in this chapter.)

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 or an execution starting point. Desk accessories and drivers, by contrast, don't require a main program, but 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 "Segment Loader" chapter of *Inside Macintosh.*

Linking

This section describes how to link an application, MPW tool, desk accessory, or driver.

❖ *Note:* For more information about Linker functions, see "More About Linking" in this chapter. The Link command itself is described in Chapter 9. The MPW object-file format is described in Appendix H.

The Linker links object files into an application, MPW tool, desk accessory, driver, or other executable resource. The Linker's output is an executable object file. The Linker 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) or a new one (Figure 7-2).

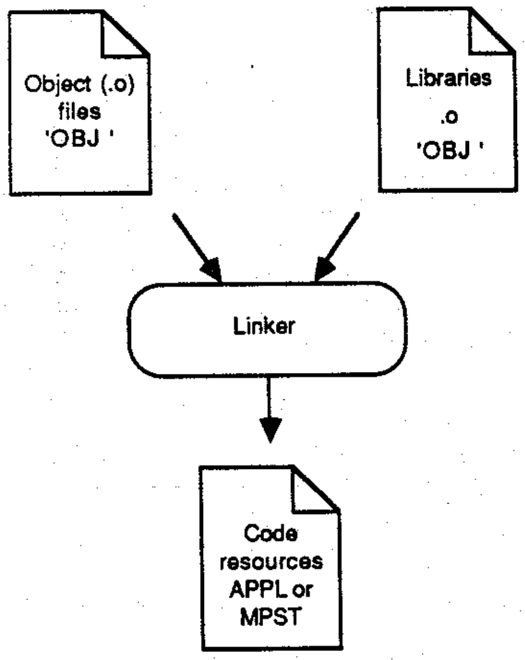


Figure 7-2
Linking

The Linker resolves all symbolic references, and also controls final program segmentation. A related tool, Lib, provides facilities for modifying and combining object files (libraries).

The Linker'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" below.)

What to link with

Applications, tools, and desk accessories should be linked with the libraries listed in Table 7-1. It's wise to link new programs with all of the libraries that might be needed. If unnecessary files are specified, the Linker will display warnings indicating that they can be removed from your build instructions.

Table 7-1
Files to link with

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. You may also call routines in the following 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 tools and desk accessories, refer to "Linking a Tool" and "Linking a Desk Accessory or Driver" later in this chapter.

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 get linked with your program. (You can use the Linker's **-w** 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, the object file containing your main program must appear *before* CRuntime.o in the list of object files passed to the Linker.

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 command files and are interpreted by the Shell. An attempt to run a file of any other type produces an error message. Table 7-2 summarizes file types and creators.

Table 7-2
File types and creators

Type of Program	Type	Creator
Application	APPL	*any*
MPW tool	MPST	'MPS '
Desk accessory	DFIL	DMOV
Command file	TEXT	*any*

❖ *Note:* Each application has its own unique creator (or **signature**)—see the "Finder Interface" chapter of *Inside Macintosh.*

You can set a file's type and creator with the **-t** and **-c** options to Link, Rez, or SetFile.

Putting together an MPW tool

Typically, when a program is run on the Macintosh, it takes over the screen, puts up its own menus, and replaces the previous program. Programs with this behavior (such as MacPaint or MacWrite) are called applications. All of the programs previously available on the Macintosh, except for desk accessories, fall into this category.

The Shell also provides an environment for a new type of program called an **MPW tool**. Tools are similar to desk accessories in many aspects of their behavior. When a tool is run from the Shell, it does not replace the Shell nor erase the screen, but instead runs within the Shell environment and has access to the facilities provided by the Shell. The Assembler, the Compilers, Link, Make, and so on are all tools in the MPW system.

For a description of the facilities available to an MPW tool, see Appendix F, "Writing an MPW Tool."

Linking a tool

Linking an MPW tool is the same as linking an application, 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 files Stubs.a or Stubs.c— these files contain dummy library routines that are used to override standard library routines that aren't used by MPW tools, thus reducing the tool's 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.)

Putting together a desk accessory or driver

A **desk accessory** is a 'DRVR' resource whose resource name begins with a null character ($00), and that resides in the System file. To make it convenient to write a desk accessory or driver in Pascal or C, MPW provides the following:

- The library DRVRRuntime.o, which contains the glue for the driver routines *open*, *prime*, *status*, *control*, and *close*.
- The resource type 'DRVW', declared in :RIncludes:MPWTypes.r. The 'DRVW' resource is a special case of a 'DRVR' resource, and contains constants that point to the addresses of the driver routines in DRVRRuntime.o.

For information on writing a desk accessory using the 'DRVW' resource and DRVRRuntime routines, see Appendix G, "Writing a Desk Accessory or Other Driver Resource." The remainder of this section describes how to put together a desk accessory and install it.

Putting together a desk accessory or driver requires two steps:

1. Link your driver code with the DRVRRuntime library 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.
2. Use the Resource Compiler, Rez, to create the final driver file. That is, you'll need to 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'll then need to install your desk accessory in the System file by using the Font/DA Mover.

Figure 7-3 illustrates the process of building a desk accessory or other driver.

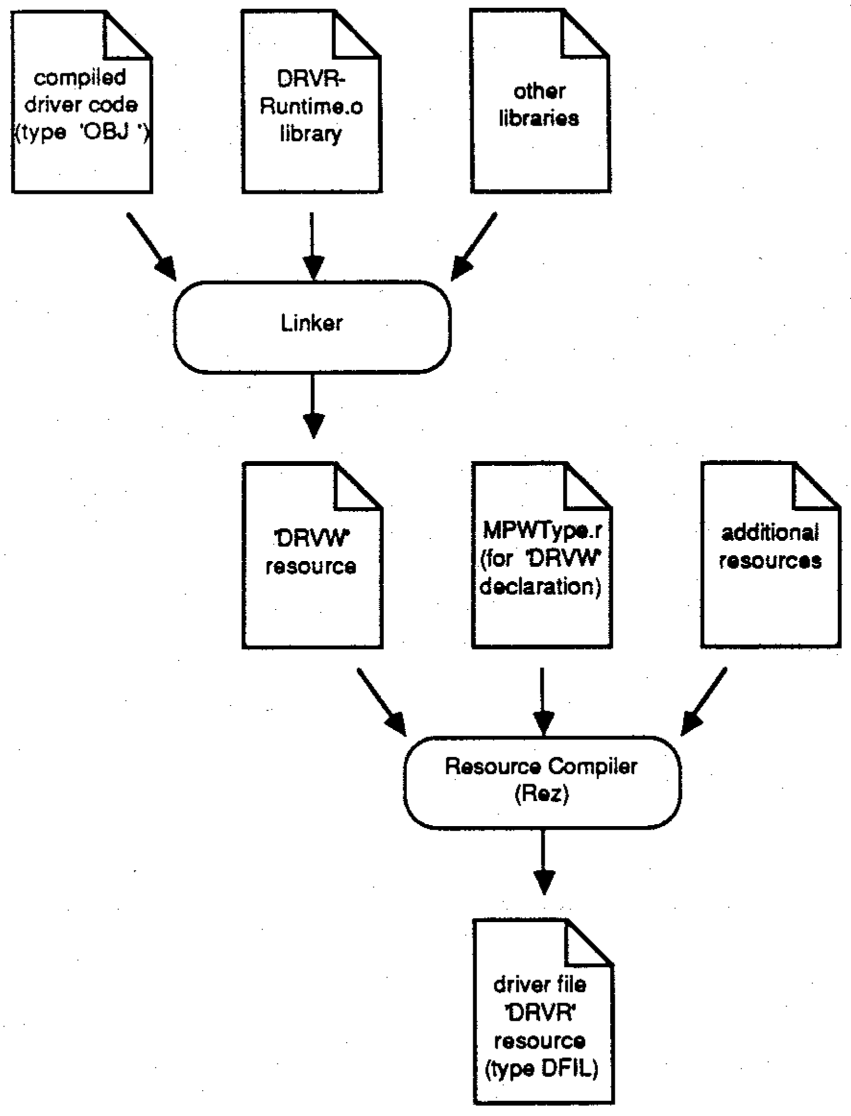

compiled driver code (type 'OBJ ')
DRVR-Runtime.o library
other libraries
Linker
'DRVW' resource
MPWType.r (for 'DRVW' declaration)
additional resources
Resource Compiler (Rez)
driver file 'DRVR' resource (type DFIL)

Figure 7-3
Building a desk accessory with DRVRRuntime

❖ *Note:* Of course, it's still possible to create a desk accessory directly in assembly language, without using DRVRRuntime.

Linking a desk accessory or driver

Linking a driver requires the following:

- The Linker'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.)
- When you link a desk accessory or driver, 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   -w  ∂
       -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 does the following:

- 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 "Resource Manager" chapter 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 will be overridden by the DRVRRuntime.o entry point. (A Linker warning will call attention to this.) The main entry point in CRuntime.o cannot be used for desk accessories.

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 7-1. 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 the construction of 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 is irrelevant */

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,             /* Desk accessories only do 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 "Device Manager" chapter 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 (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.

Using Make

The Make tool enables you to keep track of all of the components of a program and their relationships to each other—then, when one component of a program is updated, Make lets you automatically update all other parts of the program that depend upon 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 Chapter 9.)

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** which must exist or be up-to-date before the target can be rebuilt. For example, an application would depend upon 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 makefile can include dependency rules, variable definitions, and comments. Table 7-3 summarizes the syntax of a makefile, and the sections following the table go into more detail.

Table 7-3
Makefile summary

targetFile... f [*prerequisiteFile...*] [*ShellCommands...*]	Dependency rule, with or without build commands (*f* is Option-F on the keyboard). This rule means that *targetFile* depends upon *prerequisiteFile*. If any of the prerequisites are newer than the target the subsequent Shell commands are output. *Important:* Build commands must begin with a space or tab.
targetFile... ff [*prerequisiteFile...*] *ShellCommands...*	Dependency rule, requiring its own set of build commands
.[*suffix*] *f .suffix* *ShellCommands...*	Default rule (specifies suffix dependencies)
targetDirectory: ... *f 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:* Makefile input lines may not exceed 255 characters.

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       ƒƒ      Sample.r        # Sample depends on Sample.r
   Rez Sample.r -o Sample

Sample       ƒƒ      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 folders for each of the MPW languages (introduced in Chapter 1).

Dependency rules

A **dependency rule** specifies the component (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 of the prerequisite files is newer than the target file. 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. Make looks at the modification dates of the prerequisite files (and their prerequisites, if any) and decides whether the target needs 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.)

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.

More than one target file name 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-hand 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 that target. The built-in Make variable {Targ} has the value of the current target.

You can also state multiple dependency lines for a single target—multiple dependencies mean that the target depends on all of the prerequisite names that appear on all of the lines. If no build commands are specified for a dependency line, the build commands are taken from other dependency rules, or default rules, if present.

❖ *Note:* With the standard "single-*f*" form of the dependency rule, only one sequence of build commands may be specified for any given target. Thus, on dependency rule specifies a target's build commands, and additional rules simply specify additional dependencies.

Double-*f* dependency rules

Double-*f* dependency rules are slightly different from the standard single-*f* rules. Syntactically, a double-*f* dependency entry is the same as a single-*f* entry, except that ***ff*** is used in place of ***f***. The difference is that each double-*f* rule requires its own set of build commands. For example,

```
TargetFile   ff  A  B
      build commands-1
TargetFile   ff  C  D
      build commands-2
```

If the target is out-of-date with respect to one or more dependency set, each of the corresponding sets of build commands will be output. 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.)

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.)

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]  f  .suffix2
        ShellCommands ...
```

(Note that the period must be present for a default rule to be recognized. The period is taken as part of the suffix.)

The power of default rules is that many specific dependencies and build commands can be expressed by a single rule. Make has built-in default rules for assemblies and for C and Pascal Compilers. You need to specify only the dependencies not covered by default rules.

Default rules are applied only when no build commands have been given for a particular target. You can override the built-in default rules by placing your own versions of the default rules in the makefile. You can augment the default rules for a particular file by additional dependency rules, as long as these dependency rules do not include build commands.

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-hand 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 dependencies may result from other units that you use or refer to in your file—these may be include files, C header files, or Pascal USES files.)

The data fork of Make 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 default-rule build commands by defining these variables in the makefile. (For instance, you might want to specify the location of your include files by adding a **-i** *pathname* option).

You can redefine these 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" below for information.

{DepDir} and {TargDir} are built-in Make variables that allow default rules to work in the same or different 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 following section.

{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).

❖ *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.

Directory Dependency Rules Normally, default rules work only within a single directory. 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: ... *f* *searchDirectory*: ...

Directory dependencies are identified by dependency names that end in colons (that is, directory names). For example,

```
ObjFiles:   f   SrcFiles:
```

The above rule, together with the standard default rules, would mean, for example, that ObjFiles:***name***.c.o depends upon SrcFiles:***name***.c. 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-hand side of the rule. The names are tried in the order that they appear in the rule.

❖ *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.

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.

If Make doesn't recognize a variable reference in a build command line, it is left unchanged, so that it can be processed later by the Shell. (Unidentified variables in dependency lines are reported as errors.)

Caution

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*} will be replaced by *stringValue*. One common use of variables is to parameterize the directory portion of filenames so that you can easily adapt a makefile to different directory setups.

❖ *Note:* Make variables in build command lines are not expanded until Make generates the build commands—because command generation follows all dependency rule processing, there's no requirement that variable definitions appear before their references in build command lines.

You can define a variable on the command line with Make's **-d** option; this overrides any definition within the makefile.

Built-in Make variables

The following built-in Make variables have values that are dynamically assigned as Make generates the build commands:

{Targ}	The complete filename of the target on the left-hand 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.

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 quoting 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 quotes are removed.
"..."	Quotes the enclosed string, but {...} variable references are expanded, and the escape character ∂ is processed. The double quotes are removed.

Quotes are processed as follows:

- In dependency lines and in the name part of variable definitions, quotes literalize the quoted characters (useful for file or variable names).
- On the right-hand 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 with ∂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 command file. The simplest form of such a command file consists of the two commands:

```
Make {"Parameters"} > MakeOut
MakeOut
```

The first command executes Make, using the parameters passed to the command file. (See the description of the {"Parameters"} variable in Chapter 3.) Output (that is, build commands) is redirected to the file MakeOut. The second line of the command file executes MakeOut.

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 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, thus, are 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 due to 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" is introduced; that is, unexpected behavior may occur when generated commands alter the assumptions that Make used to determine whether targets were out-of-date. (You're not likely to 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 f .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 the Make tool itself. A series of explanatory notes follows the listing.

```
#################### Variables #####################
```

```
ToolDir         =       {Boot}ToolUnits:                                     #SEE  NOTE  (1)
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  =   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 (duplicates due to Stubs.a.o)
                                                                     #SEE  NOTE  (3)

SourceFiles     =       Make.p                        ∂
                        DefaultRules                  ∂
                        Makefile

########### Default Rule Customizations ############

POptions        =       -i {Boot}ToolUnits:                          #SEE  NOTE  (4)

################ Dependency Rules #################

MakeX                   ƒƒ                  {MakeObjs} {Libs}        #SEE  NOTE  (5)
        Link {LinkOpts} -p -b -o MakeX                ∂
        -t MPST -c "MPS "                             ∂
        {MakeObjs} {Libs} ≥LinkMsgs

MakeX                   ƒƒ                   defaultRules
        Duplicate -d defaultRules MakeX -y

MakeX                   ƒƒ                   {MakeObjs} {Libs} defaultRules
        SetFile MakeX -m . -d . #set last-mod and creator dates

Make.p.o                ƒƒ                   {MakeUses}                  #SEE NOTE (6)
        Delete MakeLoad -i        #delete Make's Load/Dump file if out-of-date

Make.p.o                ƒƒ                   Make.p
        Save Make.p ≥Dev:Null || Set Status 0    #save source before compile if changed

Make.p.o                ƒƒ                   {MakeUses}  #will be augmented by default rules

{ToolDir}MacInterfaces.p.o  ƒ                {PInterfaces}MemTypes.p     #SEE NOTE (7)∂
                                             {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 (8)
        Duplicate -y ≈ MakeSrc:                   #backup to Sony

Restore                 ƒ
        Duplicate -y MakeSrc:≈ :                  #restore from Sony

Listings                ƒ                    {SourceFiles}               #SEE NOTE (9)
        Print -h -r -ls .85 -s 8 -b -hf helvetica -hs 12 {NewerDeps}
        Echo "Last listings made `Date`" >Listings
```

Notes on Make's makefile

(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.

(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) The {POptions} definition gives a value to one of the variables used in the default rules, customizing it for this particular makefile.

(5) The three sets of *ƒƒ* rules for MakeX handle (a) the Make link, (b) the copying of the default rules to Make's data fork, and (c) the setting of the creation and modification dates. The link will take place only if the Make objects or libraries change. The default rules will be copied only if the rules have changed. And the setting of the dates will take place if either of the first two rules was activated. (Note that the third rule has the union of the dependency relations of the first two.)

(6) The three sets of *ƒƒ* rules for Make.p.o control the compilation of the main source for Make, with some interesting side effects. The Make source uses the Pascal Compiler's $LOAD option, which creates a symbol table for the USES that can be loaded much faster than the USES are normally processed. The first of the *ƒƒ* rules is used to delete this load file (MakeLoad) if it has been invalidated by a change in the USES files. This rule is interesting in that it deletes rather than builds something. The second of the *ƒƒ* rules saves the Make source before it is compiled, only if the source file has changed. The last of the *ƒƒ* rules 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.

(7) 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.

(8) 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 Make itself. (The Make program is represented by the MakeX target—MakeX standing for experimental version of Make.) Note that the Backup and Restore targets do not actually get built by their build rules; thus they are "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.

(9) 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 the Echo command last wrote the date into the Listings file. 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. (The fact that the date is written 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 will be 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 will be generated, if necessary, before the link.

More about linking

This section supplements the information given under the description of the Link command in Chapter 9 and earlier in this chapter under "Linking." This section may be of interest after you're familiar with the major MPW tools and are ready to optimize your programs or build procedures.

Linker 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 the Linker'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 "Segment Loader" chapter 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 the jump table will be 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.
- 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 (**-l** option).

The Linker 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.

❖ *Note:* If Linker errors or a user interrupt cause the output file to be invalid, then the Linker sets the file's modification date to "zero" (Jan. 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 unused parts of a program to be unloaded and purged from memory, thus freeing up memory space. You specify the beginning of a segment by placing a directive in your program's source file—see the appropriate MPW language reference manual for information. Each segment is linked into a 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—the Linker collects code for a given segment name from all of the Linker input files and places it into a single segment in the output file.

Caution

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 DumpObj command with the **-n** (names) option.

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 32760 bytes. This limit ensures compatibility with all versions of the Macintosh ROM. Larger segments are allowed with the Linker'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 68000 instructions.

For more information about segmentation, see the "Segment Loader" chapter of *Inside Macintosh.*

Segments with special treatment

When linking a main program, the Linker 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 to avoid memory fragmentation. This can be done by calling UnloadSeg with the address of entry point _DATAINIT as its parameter; for example,

  ```
  UnloadSeg(&_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.

Setting resource attributes

Resources have attributes that control when and how they are loaded. The default resource attribute values set by the Linker are shown in this table:

'CODE' resource	Resource attributes		
	hex	*decimal*	
0 (JumpTable)	$20	32	resPurgeable
1 ("Main")	$34	52	resPurgeable+resLocked+resPreLoad
others	$20	32	resPurgeable

❖ *Note:* For linking MPW tools (programs with output file type MPST and output file creator 'MPS '), all segments default to resPurgeable. Make sure that you *do not* set the resLocked bit for a tool.

The Linker option **-ra** sets the resource attributes. Some useful resource attribute values are

$20	32	resPurgeable
$10	16	resLocked
$08	8	resProtected
$04	4	resPreLoad

For more information about resource attributes, see the "Resource Manager" chapter of *Inside Macintosh.*

The Linker also sets the resChanged attribute (when a changed resource is in memory, and needs to be updated in the file). The Linker does not check or enforce settings for the other resource attribute bits, with one exception: The Linker *does not* support the "system heap" attribute,

$40	64	resSysHeap (for drivers, and so on)

and forces it to zero.

❖ *Note:* If you need the resSysHeap bit to be set, you'll have to process the file after the link is completed, using either Rez or ResEdit. For example, to set this bit for a 'DRVR' resource with ID=14 and the name ".printer", you could use the following command in a Rez input file:

```
include "Link.Out" 'DRVR'(14) as 'DRVR'(14,".printer",sysheap);
```

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.

The Linker 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 ∂
      etc.
```

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.

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, "Object File Format."

Symbols in object files are either 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 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.

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.

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 Linker option **-ma** (module alias). Whenever the Linker encounters an unresolved symbol, it checks the list of module aliases in an attempt to resolve it.

Linker location map

If you specify the Linker option **-l**, the Linker writes a **location map** to standard output. The map is produced in location ordering, that is, sorted by *segNum, segOffset*.

The format is divided into the following fields:

name *segName* *segNum,segOffset* [*@JTOffset*] [#] [E] [*fileNum,defOffset*]

For example,

```
 seg Main 1
TEFROMSCRAP    Main           1,422                                   2,12892
TETOSCRAP      Main           1,444                                   2,12946
%_BEGIN        Main           1,46C                                   3,3398
%_INIT         Main           1,46E                                   3,3420
```
etc.
```
 size Main 7CA

 seg %GlobalData 0
#0001          %GlobalData    0,0                    #                3,2332
__PASHEAP      %GlobalData    0,C                                     3,2886
PASHEAP        %GlobalData    0,30                   E                3,2892
QUICKDRAW      %GlobalData    0,FE                   E                4,4826
_SAGlbls       %GlobalData    0,FE                   #                4,4834
```
etc.
```
 size %GlobalData 26C
 seg %A5Init 3
_DATAINIT       %A5Init         3,0                                                  4,6338
_DataInit       %A5Init         3,0             @32             E                    4,6558
#0001           %A5Init         3,C8                            #                    4,6574
_A5Init         %A5Init         3,C8                            E                    4,6586
 size %A5Init C8
```

- *JTOffset* is a hex number giving the distance from the memory location pointed to by register A5 to the jump-table entry of the symbol.
- The number sign (#) indicates a local symbol, that is, not necessarily a unique name.
- The symbol "E" indicates an entry point in the immediately preceding module.
- *FileNum* and *defOffset* are hex numbers giving the file number and offset within the file where the symbol is defined. They are included only if the **-lf** option is also specified.

Fields in the Linker listing are separated with one or more tabs. To align the output as a table, set the output file's tab setting to 10—this gives 20 characters for the name field, and 10 characters for the other fields.

The map of static global variables is presented as a data segment named %GlobalData. The offsets in this segment are positive—the zero byte is furthest below A5, and the highest-offset byte is the byte immediately below A5. In order to translate these positive offsets into negative offsets from A5 (as shown by the debugger), you need to subtract the size of %GlobalData from the offset.

No map information is provided for the data initialization descriptors, which are appended to segment %A5Init.

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 the Linker's performance:

- **Use a RAM cache.** The Linker must open and close many object files (particularly with the **-bf** option). Experience has shown that large links run up to four times faster when you use a RAM cache of 64K or more. (Use the Control Panel desk accessory to check your RAM cache settings—if you change the setting, you must restart the MPW Shell before the new setting takes effect.) Don't use the RAM cache on machines with only 512K of memory.
- **Use the Lib utility to combine input files.** You can use the Lib command to reduce the number of input files so that the **-bf** option is not needed. This can give a 10–15% improvement in link speed (and even more on a Macintosh XL with many files). See "Library Construction" later in this chapter.

- **Eliminate unneeded files.** You can eliminate unneeded input to the Linker by heeding the warnings "File not needed for link," and deleting the files that are so identified. 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 Linker speed by as much as 40%. See "Library Construction" below for information.

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 Lib tool and its options are described in Chapter 9. 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 guarantees that the output of a link using the output of Lib is the same as a link using the "raw" files produced by the Compilers and Assembler.

Object files that have been processed with Lib result in significantly faster links than the "raw" object files produced by the Compilers and Assembler. The reasons for the speed improvements are:

- 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; 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, "Macintosh Workshop Files.") 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

- rename external modules (with the **-mn** option)
- 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 detailed 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, "Object File Format," in order 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 the Linker's progress information. (Use the **-p** option.) The Linker 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 which your particular program doesn't require.

Unreferenced modules can be removed in three steps:

1. Use the Linker'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 which 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 immediately rebuild the specialized library. You can simply include the newer version of the object file in the link list, 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 the Linker. Later, after the file is stable again, you can rebuild the library.

Chapter 8

Debugging With MacsBug

About MacsBug xx

Installing MacsBug xx

Theory of operation—a technical aside xx

The Boot Process x
Memory Usage x
MacsBug Exceptions x

Using MacsBug xx

The MacsBug command language xx

Numbers x
Strings x
Symbols x
Expressions x
Commands x

General commands xx

Memory commands xx

Break commands xx

A-trap commands xx

Heap zone commands xx

Disassembler commands xx

MacsBug summary xx

This chapter describes the theory and operation of MacsBug, the Macintosh 68xxx debugger. It also describes the syntax of commands accepted by MacsBug.

About MacsBug

MacsBug is a line-oriented, single-Macintosh debugger. It resides in RAM along with the program being debugged. The capabilities of MacsBug include

- displaying and setting memory and registers
- disassembling memory
- stepping and tracing through both RAM and ROM
- monitoring system traps
- displaying and checking the system and application heaps

MacsBug obtains control when certain 68000 exceptions occur. You can then examine memory, trace through the program, or set up break conditions and execute the program until those conditions occur.

MacsBug works with the following hardware configurations:

- 512K to 4 Mbytes of RAM
- 64K or 128K (Macintosh Plus) ROMs
- Motorola 68000 or 68020 processors
- Motorola 68881 floating-point coprocessor

❖ *Note:* MacsBug does not work on the Macintosh XL. Macintosh XL users should use MacsBug.XL, which is also provided.

Installing MacsBug

MacsBug is not a normal Macintosh application. Instead, MacsBug installs itself once at boot time and remains active until shutdown. Installation occurs if the following conditions are met:

1. MacsBug exists and is named "MacsBug".
2. It is on a startup (bootable) disk.

3. It is in the current System Folder. (This is a requirement only on HFS volumes.) The System Folder is, by definition, the folder that contains a system file named System and a system file named Finder.

MacsBug is shipped in the Debuggers folder; you must move it to the System Folder to install it.

After a successful installation, the message "MacsBug installed" is displayed below the "Welcome to Macintosh" message. The startup application (normally the Finder) is then launched as usual.

Once MacsBug is installed, the only way to remove it is to reboot. To prevent the installation of MacsBug during a boot, hold the mouse button down while booting. To permanently override the installation of MacsBug, simply rename it or remove it from the disk.

❖ *Using HFS with the 64K ROM:* If you have MacsBug on an HD-20 (HFS) Startup disk on a machine with the original 64K ROM, there is a seeming conflict with the above mouse-down command, because holding the mouse button down at boot also forces the Macintosh to boot from the floppy disk rather than switch-launching to the HD-20. However, a skillful mouser can accomplish either command simply by knowing that MacsBug is installed first, right after the "Welcome to Macintosh" hello message, and that the HFS code looks for a mouse-down only after the "MacsBug installed" message.

Theory of operation—a technical aside

This section provides background information about how MacsBug works. Some of this information is important if you are interested in implementing your own Macintosh debugger; most other readers can skip this section.

The boot process

The state of the world when MacsBug is about to be loaded is fairly complete. The interrupt system, Memory Manager, and ROM-based I/O drivers have already been initialized by the ROM boot code. The boot code initializes the Event Manager, the Font Manager, the Resource Manager, and the file system. (Although the Toolbox is initialized at this point, MacsBug does not use the Toolbox.) The 'DSAT' table is loaded in and the string "Welcome To Macintosh," contained therein, is displayed.

Next, the loading process of MacsBug takes place as follows: First the boot-blocks code reserves some space (1024 bytes) for MacsBug's own global variables. Then this code looks for the file specified in the boot blocks, as described above. If the file is not found, then the global space is deallocated and the boot process continues normally without installing a debugger.

If MacsBug is found, the data fork (not the resource fork!) of the file is loaded onto the current stack, which is located immediately below the main screen buffer in memory.

❖ *Historical note*: For reasons relating to the original Lisa Workshop, the first block (512 bytes) of the MacsBug data fork is stripped off during this loading process.

The boot code then JSRs to MacsBug itself. MacsBug begins its installation process by checking to see if the mouse button is down. If it is, MacsBug aborts the installation and lets the boot process continue without installing itself. If the button is not down, MacsBug determines which kind of machine and microprocessor it is running on, and configures itself accordingly.

At the successful completion of the installation, the message "MacsBug installed" is posted below the "Welcome To Macintosh" screen. The boot process then continues by loading 'INIT' resources from the System file.

❖ *Technical note*: The boot code looks for 'INIT' resources 0-31 and JSRs to them. These 'INIT' resources are used to set up the keyboard maps ('INIT's 0 and 1), install patches (of type 'PTCH') to ROM code, and so on. 'INIT' 31 extends the system further by looking for any files of type INIT in the System Folder. This facility allows you to install your own startup code without changing the System file.

Finally, the startup application is launched. The startup application is typically the Finder, but can be set to any other application via the Finder's "Set Startup" menu item.

Memory usage

During installation, MacsBug obtains further memory from below the main screen buffer for use as its own screen memory. (MacsBug obtains memory from the same general area in RAM as do RAM disks and caching utilities, based on the location of BufPtr, a Macintosh global variable.) MacsBug offers a full screen display with 40 lines saved. This display uses about 20K of memory.

The total RAM requirement of MacsBug is approximately as follows:

Global space	1 K
Screen space	20 K
Code space	24 K
TOTAL:	45 K

MacsBug may not work with some memory-intensive applications on a Macintosh 512K. For example, using MacsBug and the MPW Pascal Compiler on a Macintosh 512K severely limits the size of programs that may be compiled. Two solutions are possible:

1. Remove MacsBug to free up about 45K of RAM.
2. Add additional RAM via the Macintosh Plus Logic Board Upgrade or a compatible third-party hardware upgrade to 1 MB or more of RAM.

MacsBug exceptions

When installed, MacsBug puts pointers to itself in many of the hardware exception vectors in addresses $0000 0000 through $0000 00FF. It then remains dormant until one of "its" exceptions occurs. The following is the list of exceptions to which MacsBug responds; each is numbered one greater than the corresponding Macintosh System Error number.

Exception #	**Assignment**
2	Bus Error (rarely seen on the Macintosh)
3	Address Error (not aligned to a word boundary)
4	Illegal Instruction (bit pattern not recognized)
5	Zero Divide
6	CHK Instruction (array index out of bounds)
7	TRAPV Instruction (overflow)
9	Trace (used to single step in MacsBug)
10	Line 1010 Emulator (the A-trap handler for all Toolbox traps)
11	Line 1111 Emulator (68xxx coprocessor trap interface)
28	Level 4 Interrupts
29	Level 5 Interrupts
30	Level 6 Interrupts
31	Level 7 Interrupts
47	Trap $F Instruction (used for setting programmer breaks)

68000 exception processing is described in the Motorola 68000 *Programmer's Reference Manual.*

Any time an A-Trap or other exception listed above occurs, MacsBug intercepts the trap and can thus stop or display the current state of the machine. Single-stepping through 68xxx instructions is possible because MacsBug can set the Trace bit in the status register of the microprocessor. MacsBug saves the ROM-based A-trap handler address in the long word immediately preceding its own A-trap handler routine. Thus, if you need to access the real ROM A-trap handler when MacsBug is installed, you can look at the long word before the address of the current handler.

Using MacsBug

The simplest way to get into MacsBug is to generate an exception by pressing the interrupt button. (The interrupt button is the rear button of the programmer's switch on the Macintosh, or the minus key on the numeric keypad on the Macintosh XL.)

To see the application screen while the debugger is active, press the tilde/back quote key (~/ `) in the upper-left corner of the keyboard. To restore the debugger's display, press any character key. Repeated presses toggle between the two screens, allowing easy viewing of both the actual code (MacsBug screen) and the results (main screen).

The best way to enter the debugger programmatically is to set a breakpoint in your program by using the system trap called Debugger at the point where you want MacsBug to get control. There are two ways to use this trap. Calling trap $A9FF drops into MacsBug and displays the message "USERBRK". It then does a normal exception entry into MacsBug (unless you have toggled the DX command—see "Break Commands" below).

If you want to display custom debugging information, declare and call the trap with bit 10 set ($ABFF). When this latter trap is encountered, MacsBug assumes that the top of the user's stack has a pointer to a Pascal string. It prints out the string, displays the message "USERBRK," and does a normal exception entry into MacsBug. As $ABFF is a procedure call, MacsBug takes care of popping the string pointer off the stack.

Here is a summary of how to declare and use this trap on a per language basis.

Assembly language:

Declaration:

```
    _Debugger      OPWORD      $A9FF       ; predefined in the file ToolTraps.a
    _DebugStr      OPWORD      $ABFF       ; not predefined - define yourself
```

Example calls:

```
a)  _Debugger                              ; enters MacsBug and displays USERBRK

b)  STRING PASCAL                          ; Asm directive to make sure to push a
                                           ;  Pascal string
    PEA #'Entered main loop'               ; push address of string on stack
    _DebugStr                              ; enters MacsBug and displays message
```

Pascal:

Declaration:

```
    PROCEDURE Debugger; INLINE $A9FF;
    PROCEDURE DebugStr(str: str255); INLINE $ABFF;
```

Example calls:

```
a)  Debugger;                               {enters MacsBug and displays USERBRK}
b)  DebugStr('Entered main loop');          {Enters MacsBug and Displays message}
```

MPW C:

Declaration:

```
    pascal void Debugger() extern 0xA9FF;
    pascal void DebugStr(aString) Str255 aString; extern 0xABFF;
```

Example calls:

```
a)  Debugger();                               /*enters MacsBug and displays USERBRK*/
b)  DebugStr(c2pstr("Entered main loop"));    /*enters MacsBug, displays message*/
```

When MacsBug gets control, it disassembles the instruction indicated by the program counter and displays the contents of the registers. If the exception was caused by a $A9FF or $ABFF instruction, MacsBug displays the message "USERBRK", advances the PC to the next instruction, and then disassembles the instruction and displays the registers. It then displays the greater-than symbol (>) as a prompt, indicating that it is ready to accept a command.

❖ *Note:* There are two other ways to enter MacsBug: by using 'FKEY' and 'INIT' resources. With ResEdit, a skilled user can create a custom resource of either type whose sole function is described by two simple 68000 instructions: `$A9FF $4E75` (`_Debugger` and `RTS`; that is, the sequence to enter MacsBug).

Warning

Another way to generate an exception that was popular in the past was to add a line such as

```
        DC.W  $FECE          ; generate a line 1111 exception
```

at the point in your program where you wanted MacsBug to get control. (Any value $F000 through $FFFF could have been used.) *This method should not be used any more*, as these instructions have been reserved by Motorola for use in their coprocessor interface for the 68020 microprocessor. (For example, in the future these "exceptions" could actually be MC 68881 floating-point instructions!)

The MacsBug command language

Commands consist of a one- or two-character command name followed by a list of zero or more parameters (depending on the command). A return character repeats the last command entered, unless otherwise specified in the command description.

Parameters can be numbers, text literals, symbols, or simple expressions. All parameters can be entered as expressions. Parameters are represented by descriptive words and abbreviations such as ***address***, ***number***, and ***expr***.

MacsBug commands can be divided into five groups: memory, break, A-Trap, heap zone, and disassembly commands.

Numbers

As is fitting for a debugger, ***all numbers are hex unless otherwise specified.*** Decimal numbers are preceded by a number sign (#). Hexadecimal numbers can optionally be preceded by a dollar sign ($). Numbers can be signed (+ or –). A hex word (four hex characters) preceeded by a less-than symbol (<) is sign-extended to a long word.

Here are some numbers in different formats—the formats shown are the same as those displayed by the CV (Convert) command, described later in this chapter.

Number	Unsigned hex	Signed hex	Decimal
$FF	$000000FF	$000000FF	#255
37	$00000037	$00000037	#55
-FF	$FFFFFF01	-$000000FF	-#255
#100	$00000064	$00000064	#100
+10	$00000010	$00000010	#16
#-32	$FFFFFFE0	-$00000020	#-100
<FFFA	$FFFFFFFA	-$00000006	#-6

Strings

A text literal is a one- to four-character ASCII string bracketed by single quotes ('). If a string is longer than four characters, only the first four characters are used. When used by MacsBug, text literals are right justified in a long word. Here are some examples:

String	Stored as
'A'	$00000041
'Fred'	$46726564
'1234'	$31323334

Symbols

The following symbols are generally used to represent the 68xxx registers:

`RA0, RA1,...RA7`	the contents of address registers A0 through A7
`RD0, RD1,...RD7`	the contents of data registers D0 through D7
`PC`	the contents of the program counter
`SR`	the contents of the status register

❖ *Note:* In any expression where you want to use the value of one of the main registers, use the R*xx* form, as shown above. If you specify A0 for example, it will be interpreted as address `A0`, a valid hex address, not as register A0.

In addition, the following symbols are used for frequently referenced locations:

.	a period ("dot") gives the last address referenced
`TP`	"thePort"—the address of the current QuickDraw port

Expressions

Expressions are formed by operators acting on numbers, text literals, and symbols. The operators are

+	Addition (infix); assertion (prefix)
–	Subtraction (infix); negation (prefix)
@ or *	Indirection operator (two different prefix operators with identical functionality)
	Note: The indirection operator uses the long integer at the location pointed to by the operand.
&	Address operator (prefix)
<	Add sign-extended number (infix); sign extension (prefix)

Expressions are evaluated from left to right. All operators are of equal precedence. There is no way to alter the order of evaluation. Here are some valid expressions:

```
RA7+4
3A700-@10C
TP+#24
-RA0+RA1-'FRED'+@@4C50
RA5<FE34
```

(`RA5<FE34` is the same as `RA5+FFFFFE34`—useful in looking at global variables.)

General commands

?

(Help). Displays a short list of MacsBug commands and their parameters.

DV

(Display Version). Displays the version, the date and time of creation, and signature of MacsBug. For example,

```
MACSBUG 5.1B1  17-May-86 00:05:10  <DKA>
```

RB

(Reboot). Reboots the system.

ES

(Exit to Shell). Invokes the trap ExitToShell, which causes the current shell to be launched. (The "current shell" is usually the Finder but can be changed by editing the Finder field of the boot blocks.) The current shell must reside in the System Folder and is logically distinct from the startup application.

❖ *Technical note:* ES may not work with applications that override important system traps. This problem occurs because the application heap gets initialized promptly upon calling the trap ExitToShell; the initialization usually trashes any system patches that were located there. However, there is a hook called IAZNotify, called by InitApplZone, that you can use to restore the world before purging the otherwise necessary routines.

EA

(Exit to Application). Relaunches the application. This is a faster method than calling ES and relaunching from the Finder.

Memory commands

CV *expr*

(Convert). Displays *expr* as unsigned hexadecimal, signed hexadecimal, signed decimal, text, and binary.

DM [*address* [*number*]]

(Display Memory). Displays *number* bytes of memory starting at *address.*

Number is rounded up to the nearest 16 bytes. If *number* is omitted, 16 bytes are displayed. If *address* and *number* are both omitted, the next 16 bytes are displayed. The dot symbol (.)is set to the address of the beginning of the last block displayed.

If *number* is set to certain four-character strings, memory is symbolically displayed as a data structure that begins at *address*. The strings and the data structures they represent are

`'IOPB'`	Input/Output Parameter Block for File I/O
`'WIND'`	Window Record
`'TERC'`	TextEdit Record

(Refer to *Inside Macintosh* for a description of these data structures.)

You can usually terminate a DM command by pressing the Backspace key.

SM *address expr...*

(Set Memory). Places the specified values, *expr...*, into memory starting at *address*. The size of each value depends on the "width" of each expression. The width of a decimal or hexadecimal value is the smallest number of bytes that holds the specified value (four-byte maximum). Text literals are from one to four bytes long; extra characters are ignored. Indirect values are always four bytes long. The width of an expression is equal to the width of the widest of its operands. The dot symbol (.) is set to *address*.

DB [*address*]

(Display Byte). Displays a single byte of memory located at *address*. This command automatically calls the convert routine as well, allowing you to see flags easily. DB is useful for looking at the contents of memory-mapped IO registers. (Using DM will read larger portions of memory; this can have undesired side effects on the peripheral chips being examined.)

SB *address* [*expr*]

(Set Byte). Places the value of *expr* into the byte located at *address*. If no *expr* is given, then it clears the byte to zero. Like DB, this command is useful for debugging memory-mapped I/O registers.

D*n* [*expr*]

(Data Register). Displays or sets data register *n*. If *expr* is omitted, the register is displayed. Otherwise, the register is set to *expr*.

A*n* [*expr*]

(Address Register). Displays or sets address register *n*. If *expr* is omitted, the register is displayed. Otherwise, the register is set to *expr*.

PC [*expr*]

(Program Counter). Displays or sets the program counter. If *expr* is omitted, the program counter is displayed. Otherwise, the PC is set to *expr*.

SR [*expr*]

(Status Register). Displays or sets the status register. If *expr* is omitted, the status register is displayed. Otherwise the status register is set to *expr*.

TD

(Total Display). Displays all of the 68000 registers and the PC, and disassembles the current instruction that is about to be executed upon stepping, tracing, or going.

RX

(Register Exchange). Toggles the display mode so that the registers are or are not dumped during a trace command. The disassembly of the PC instruction is not affected.

CS [*address1* [*address2*]]

(Checksum). Checksums the bytes in the range *address1* through *address2* and saves that value. The checksum is an exclusive OR of the bytes in the range specified. If *address2* is omitted, CS checksums 16 bytes, starting at *address1*. If *address1* and *address2* are both omitted, it calculates the checksum for the last range specified, saves that value, and compares it to the previous checksum for that range. If the checksum hasn't changed, CS prints `CHKSUM T`; otherwise it prints `CHKSUM F`.

❖ *Note:* For checksumming memory in conjunction with A-traps, see the AS command. For checksumming after every 68xxx instruction, see the SS command.

Break commands

BR [*address* [*count*]]

(Break). Sets a breakpoint at *address*. *Count* specifies the number of times that the breakpoint should be executed before stopping the program. If *count* is omitted, the program is stopped the first time the breakpoint is hit. If *address* is omitted, all breakpoints are displayed. You can set a maximum of 8 different breakpoints.

CL [*address*]

(Clear). Clears the breakpoint at *address*. If *address* is omitted, all breakpoints are cleared.

G [*address*]

(Go). Executes instructions starting at *address*. If *address* is omitted, execution begins at the address indicated by the program counter. Control does not return to MacsBug until an exception occurs.

GT *address*

(Go Till). Sets a one-time breakpoint at *address*, then executes instructions starting at the address indicated by the program counter. This breakpoint is automatically cleared after it is hit. (GT *address* is equivalent to a BR *address* and G with the BR being cleared after it is hit for the first time.)

T

(Trace). Traces through one instruction. Traps are treated as single instructions.

If the next instruction to be executed is a JSR to a currently unloaded segment, you will see the LoadSeg ($A9F0) trap instead of the JSR. Tracing through that instruction will not work normally. If you wish to trace through the LoadSeg trap, you need to set a low-memory global at location $12D to a nonzero value. Do a `SB 12D 1` to enable tracing through the LoadSeg call. Next, Go (G). You will break at an RTS instruction. Trace once (T) to see the absolute location that you are about to jump to. Trace again and you will be at the first step of the routine that is now loaded into memory. To turn off tracing through LoadSeg calls, simply execute `SB 12D` to clear the LoadSeg low-memory flag.

S [*number*]

(Step). Steps through *number* instructions. If *number* is omitted, just one instruction is executed. Traps are not considered to be single instructions.

SS *address1* [*address2*]

(Step Spy). Calculates a checksum for the specified memory range, then does a Go; it then checks the checksum before each 68xxx instruction is executed, and breaks into MacsBug if the checksum does not match. If *address2* is omitted, SS checksums the long word at *address1*. This feature is turned off by entering MacsBug via the programmer's switch or by SS terminating when the checksum has changed.

Step Spy is very slow. Step Spy is nevertheless useful for detecting what routines are stepping on a specific place in memory. If checking memory at every A-trap is sufficient for your needs, use the AS command, described below. (The slow motion capability of SS, however, can be useful in its own right to examine how the Finder zooms windows, for example. Think of it as a tool to study graphics algorithms.)

ST *address*

(Step Till). Steps through instructions until *address* is encountered. Unlike Go Till, this command does not set a breakpoint. Thus it can be used to step through, and stop in, ROM.

MR [*offset*]

(Magic Return). When debugging, you generally trace through a program one instruction at a time. MR lets you trace through to the end of a routine instead. When you use MR, it replaces the return address that is *offset* bytes down in the stack with the *magic* address within MacsBug; then it does a Go (described above). The RTS that would have used that address returns to MacsBug instead of to the caller. MacsBug restores the original return address, and then executes the RTS as if called by the Trace command. The prompt is then displayed, ready to trace the instruction after the RTS.

MR functions according to this formula:

```
IF offset >= A6 THEN magic = offset + 4 ELSE magic = A7 + offset
```

The default *offset* is 0. This allows you to type MR RA6 when in nested subroutine calls. The usual way to use this routine is to trace until just after a JSR (return address 0 bytes down in the stack), and then do an MR. The rest of the routine is executed, and control returns to MacsBug. This command isn't repeated when you press Return; a Trace command is executed instead.

DX

(Debugger Exchange). Normally, if either the $A9FF or the $ABFF A-trap (two forms of the Debugger trap) is executed, program execution halts and the debugger is activated. DX allows you to control whether or not program execution halts. Note that the $ABFF trap will still print a string; thus with debugger entry disabled, an effect similar to that of the AT command occurs—that is, the Macintosh screen alternates between the debugger and the program. The default is to stop at Debugger traps.

A-trap commands

The A-Trap commands are used to monitor "1010 emulator" traps, used to call the Macintosh ROM. These commands take up to six parameters (*trap1, trap2, address1, address2, D1,* and *D2*). These parameters indicate which traps and other conditions should be monitored:

- *Trap1* and *trap2* specify the range of the traps. If only *trap1* is specified, the command is invoked for *trap1.* If *trap1* and *trap2* are specified, the command is invoked for all traps in the range *trap1* through *trap2.* The defaults are $A000 and $AA00.

$A000 – $A0FF	Operating system traps
$A800 – $A9FF	Toolbox traps
0 – $6F	Shortcut expressions for OS traps
$70 and greater	Interpreted as Toolbox traps

- *Address1* and *address2* specify a range of memory addresses within which traps should be monitored. The defaults are 0 and $FFFFFFFF.
- *D1* and *D2* specify the values of data register 0 within which traps should be monitored. They are treated as unsigned numbers. The defaults are 0 and $FFFFFFFF.

Thus, if no parameters are given, all traps are monitored.

A-trap commands allow two commands to take place simultaneously. The trick to using the A-trap commands is to know that there are separate flags for tracing and breaking, and that separate globals are used for storing the general trap range (GTR) and the breaking trap range (BTR):

- Any A-trap command (AA, BA, AT, AB, AS, AH, AR) sets the tracing flag. In addition, any command except AS can supply a trap range, which is always stored in the GTR variable.
- Executing an AB or BA also sets the breaking flag. It also saves the trap range you supplied in both the GTR and BTR variables.

Previously any A-trap command would clear all flags, but now only AX clears all flags. If you are a "casual" A-trap user, execute AX before executing any A-trap commands in order to avoid undesired breaks. However, for the real MacsBug power user, combined A-trap commands can be very useful.

An example of how you can use this is as follows. If you wish to view (trace) all of the file system traps called from your application but also want to break at the next Open call that you make, you would type (and in this order!):

```
>BA Open
>AA 0 17            (shorthand for file system traps)
```

The AA command is entered second so that its range overwrites the GTR supplied by the BA command. This way you can view (trace) the "wider" range of traps while breaking on the "smaller."

BA [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(Break in Application). Causes a break when the conditions specified by the parameters are satisfied and the trap is being called from the application rather than from the ROM. *Address1* and *address2* are automically set to ApplZone and BufPtr. Therefore you can use this command to get back to the application when in ROM. Simply type BA and Go. MacsBug will be entered at the next trap called by code located in the application heap. To break on ROM calls as well (or traps called from the system heap or elsewhere), use AB, described below.

AA [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(Application A-trap Trace). Traces and displays each A-trap called from the application heap without breaking if the conditions specified by the parameters are satisfied. AA continues to display A-traps until you press the interrupt button. AA allows you to monitor only the traps that the application calls, and thus can be useful for checking and measuring performance. To monitor all traps called, including calls made from inside the ROM and traps called from the system heap, use the AT command.

AB [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(A-trap Break). Causes a break when the conditions specified by the parameters are satisfied. AB without any parameters will stop at the very next trap executed anywhere by the Macintosh. To stop at the next trap called by the current application, use BA instead.

AT [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(A-trap Trace). Traces and displays each A-trap without breaking, when the condition specified by the parameters is satisfied. AT continues to display all A-Traps until you press the interrupt button. If you wish to just see the traps called by the current application, use AA instead.

For example, to see all QuickDraw calls displayed, regardless of who calls them, you could type

```
>AT A86C A8FB
```

AH [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(A-trap Heap Zone Check). Checks the heap zone for consistency just before executing each trap in the specified range. If an inconsistency is found, it displays the addresses of the two memory blocks in question.

AR [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]]

(A-trap Record). Whenever the parameter constraints are satisfied by an A-trap call, information about the call is recorded. The trap name, PC, A0, D0, and the time are always saved. If the call was for an OS trap, 32 bytes pointed at by A0 are recorded; otherwise 32 bytes pointed at by A7 (the stack pointer) are saved. To display the current saved information, type AR with no arguments.

This command is especially useful for tracking down crashes in the Macintosh ROM. For example, the command

```
>AR 0 1000 @2AA @114
```

records traps 0 through 1000 (all traps), from ApplZone ($2AA) through HeapEnd ($114), so it will record the last trap call made from anywhere in the application heap (the application's code).

AS *address1* [*address2*]

(A-trap Spy). Calculates a checksum for the specified memory range, checks it before each A-trap that is called, and breaks into MacsBug if the checksum does not match. If *address2* is not specified, AS checksums the long word at the given address. Use SS if you want the range of memory to be checked before every 68xxx instruction rather than before every A-trap only. AS is turned off by AX.

AX

(A-trap Clear). Clears all A-trap commands.

Heap zone commands

The heap zone commands act upon the current heap zone. When MacsBug is started up, the current heap zone is the application heap zone. You can set the current heap zone by using the HX command. Several commands cause MacsBug to scramble the heap zone. When MacsBug scrambles the heap zone, it rearranges all the relocatable blocks. This is useful for finding illegally used pointers to relocatable data structures.

HX [*address*]

(Heap Exchange). Sets the current heap to *address*. If no *address* is given, then HX toggles the current heap zone between the system heap zone and the application heap zone. In any case, HX displays the resulting current heap address.

HC

(Heap Check). Checks the consistency of the current heap zone, and displays the addresses of inconsistent memory blocks as well as the address of the current heap.

HS [*trap1* *trap2*]

(Heap Scramble). Scrambles the heap zone by moving relocatable blocks when certain traps in the specified range are encountered. HS always scrambles the heap zone as a result of NewPtr, NewHandle, and ReallocHandle calls. It scrambles the heap zone as a result of SetHandleSize and SetPtrSize if the new length is greater than the current length. HS is fastest if you set *trap1* to $18 and *trap2* to $2D. The heap zone is not scrambled as a result of traps other than those named above.

HD [*mask*]

(Heap Dump). *Mask* is optional. Whether or not ***mask*** is used, it displays each block in the current heap zone in the following form:

blockAddr *type* *size* [*flag* *MP_location*] [*] [*refNum* *ID* *restype*]

- *blockAddr* points to the start of the memory block.
- *type* is one of the following letters:

F	free block
P	pointer
H	handle to a relocatable block

- *size* is the physical size of the block, including the contents, the header, and any unused bytes at the end of the block.
- For handles (*type* H), *flag* is either blank if not purgeable or a P if purgeable. Then *MP_location* is displayed, which is the address of the master pointer to the relocatable block.
- The asterisk (*)marks any locked object (nonrelocatable blocks and locked relocatable blocks).
- For resource file blocks, three additional fields are displayed: the resource's reference number, ID number, and resource type. If ***mask*** is omitted, the dump is followed by a summary of the heap zone's blocks. It begins with the six characters "HLP PF", which represent the six values that follow them. These values are

H	number of relocatable blocks in the heap zone (*handles*)
L	number of relocatable blocks that are *locked*
P	number of *purgeable* blocks in the heap zone
(space)	space, in bytes, occupied by purgeable blocks
P	number of nonrelocatable blocks in the heap zone (*pointers*)
F	total amount of *free* space in the heap zone

Here is a sample summary:

```
HLP PF 0084  0004  0002  0000079E  0017  000003B4
```

Note that block counts are single words, and values representing space in bytes are long word quantities. If ***mask*** is used, the summary line displays the block counts of specific types of blocks. Possible values for ***mask*** are

'H'	relocatable blocks (handles)
'P'	nonrelocatable blocks (pointers)
'F'	free blocks
'R'	resource blocks
'XXXX'	resource blocks of type 'XXXX'

If *mask* is used, the heap summary takes this form:

CNT ### *<# of blocks of mask type>* *<# bytes in those blocks>*

You can prematurely terminate an HD command by pressing the Backspace key.

The dot address (.) is set to the last block of memory displayed by HD.

HT [*mask*]

(Heap Total). Displays just the summary line from a heap zone dump. *Mask* works just as it does with the HD command (described above).

SC

(Stack Crawl). Assumes that LINK / UNLK A6 has been religiously performed at the beginning and end of each procedure or function. (The $D+ directive in Pascal, and the **-g** and **-ga** options in C force these instructions to be performed.) The output format is as follows:

`SF` *@<stack frame location>* *<address of call to procedure>*

For example,

```
SF @0D633C  ProcName+3A
```

means that the currently executing procedure or function has its local stack frame at $D633C and was called from ProcName+$3A (which is not the return address!). If the program counter is not in the ROM, SC may not work properly.

Disassembler commands

SX

(Symbol Exchange). Determines whether or not symbols are displayed. By default, symbols are turned on. SX affects any command that takes an address. Using symbols allows you to IL or BR on a procedure or function name. For example,

```
>IL ProcName+58
```

disassembles code starting at 58 bytes (hex) into the procedure called ProcName, and

```
>BR ProcName+58
```

sets a breakpoint at the same location. (This also works for GT, ST, DM, and so on.)

When searching for symbols, MacsBug searches the current heap (set by the HX command). The heap is searched by walking through memory and looking for locked blocks of memory. Then, within locked blocks, MacsBug first looks for a LINK A6 instruction followed by a matching UNLK A6 instruction. Then MacsBug looks for either an RTS or a JMP (A0) instruction.

Immediately following one of these last two instructions should be an eight-character symbol. This symbol must be exactly eight characters long; it should be padded with blanks if it is less than eight characters. Some compilers set the high bit on the first character of the symbol, but MacsBug clears this bit. In addition, if the high bit of the second character is set, MacsBug expects a 16-character name (used mainly for method names in MacApp-generated code.) To see all of the symbols that are valid at any given moment, use the SD command (described next).

Turning symbols off is helpful for two reasons. First, every symbol lookup traverses the current heap, and therefore may degrade the speed of the disassembly. Secondly, if you prefer always seeing a dump of code in hex rather than symbols (useful when looking at ROM code, for example), turning off symbols will guarantee a hex dump of your code. This hex dump displays the main opcode word followed by two extension words which may or may not apply to the particular instruction disassembled.

SD [*address*]

(Symbol Dump). Displays a list of the procedure names that can be found in the current heap zone. The search criteria are based on looking in each block of memory whose locked bit is set. In addition, a LINK A6 and its matching UNLK A6 must be found, followed by either a JMP (A0) or an RTS. The eight-character debugging name follows. Valid debug symbols must consist of ASCII characters in the range $20–$5F (space–underscore), inclusive. This command optionally allows you to specify a starting location for the symbol dump.

DH *number*

(Disassemble Hex). Disassembles the hex byte, word, or long word input. Typing just one byte allows you to see the general class of instructions, as *number* is left-aligned in a long word padded to the right with zeros. (Typing `DH 10`, `DH 20`, and `DH 30`, for example, shows by induction that these instruction groups are the Move.B, Move.W, and Move.L classes, respectively.)

This command is useful as a poor man's assembler. For example, if you wanted to use the RESET instruction and could not remember what its opcode was, you could type `DH 4E71` as a first guess and DH would display NOP. Trying `DH 4E70` as a second guess would reveal the actual RESET instruction.

ID [*address*]

(Instruction Disassemble). Disassembles one line at *address*. If *address* is omitted, the next logical location is disassembled. ID sets the dot symbol (.) to the *address*.

If the code has symbols compiled with it via the $D+ directive in Pascal or the **-g** option in C, and symbols have been turned on with the SX command, each address is automatically displayed as a routine name plus an offset.

IL [*address* [*number*]]

(Instruction List). Disassembles ***number*** lines starting at ***address***. If ***number*** is omitted, a screenful of lines (typically 16) is disassembled. If both ***number*** and ***address*** are omitted, a screenful of lines is disassembled starting at the next logical location. This command sets the dot symbol (.) to the ***address***.

If the code has symbols compiled with it via the $D+ option in Pascal or the **-g** option in C, and symbols have been turned on with the SX command, each address is automatically displayed as a routine name plus an offset.

You can prematurely terminate an IL command by pressing the Backspace key.

F *address count data* [*mask*]

(Find). Searches ***count*** bytes from ***address***, looking for ***data***, after masking the target with ***mask***. As soon as a match is found, the ***address*** and value are displayed, and the dot symbol (.) is set to that ***address***. To search the next ***count*** bytes, simply press Return. The size of the target is determined by the width of ***data***; it is limited to 1, 2, or 4 bytes.

For example, to find a RESET instruction in a program loaded into a Macintosh Plus, you could type

```
>F CB00 EFFFF 4E70
```

where CB00 is the beginning of the application heap, EFFFF represents the length of the application heap (roughly), and 4E70 is the RESET instruction.

WH *expr*

(Where). Takes an expression, which can be a symbolic name, and displays the location of the first routine that it finds whose name matches the expression. ROM symbol names are ten-character names, and RAM symbols are eight-character names.

If *expr* is less than $AA00, this command displays the address corresponding to the trap with that ***number***. All of the following commented commands, for example, give the same result:

```
>WH EXITTOSHELL     ; full name
>WH A9F4            ; full trap word
>WH 1F4             ; shortcut
>WH 40F6D8          ; address of ExitToShell in the 128K ROM
```

Namely,

Trap Word	*Address*	*Name*
A9F4	40F6D8	EXITTOSHELL

The shortcut method of inputting trap numbers interprets $0–$6F as OS traps, and all other traps as Toolbox traps.

If *expr* is preceded by the address operator (&), then the expression is forced to be evaluated as an address. This feature is useful for examining system patches whose addresses are often less than $AA00, the default address boundary.

If *expr* is greater than or equal to $AA00 and less than RomBase, then the address is interpreted as a user routine in RAM, and a symbolic location will be displayed if possible.

If *expr* is in ROM then the trap whose code is closest to that address is displayed.

WH is useful for finding out where you were when an error occurred. If the address expression is in RAM and the WH function returns "`PRGM AT $$$$`" you can then use the command `HD 'CODE'` to list the code segments. Then, by comparing the locations of 'CODE' segments and the current PC, you can determine which segment you are in.

MacsBug summary

General commands

?	(Help)
DV	(Display Version)
RB	(Reboot)
ES	(Exit to Shell)
EA	(Exit to Application)

Memory commands

CV	*expr*	(Convert)
DM	[*address* [*number*]]	(Display Memory)
SM	*address expr...*	(Set Memory)
DB	[*address*]	(Display Byte)
SB	*address* [*expr*]	(Set Byte)
D*n*	[*expr*]	(Data Register)
A*n*	[*expr*]	(Address Register)
PC	[*expr*]	(Program Counter)
SR	[*expr*]	(Status Register)
TD		(Total Display)
RX		(Register Exchange)
CS	[*address1* [*address2*]]	(Checksum)

Break commands

BR	[*address* [*count*]]	(Break)
CL	[*address*]	(Clear)
G	[*address*]	(Go)
GT	*address*	(Go Till)
T		(Trace)
S	[*number*]	(Step)
SS	*address1* [*address2*]	(Step Spy)
ST	*address*	(Step Till)
MR	[*offset*]	(Magic Return)
DX		(Debugger Exchange)

A-trap commands

BA	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(Break in Application)
AA	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(Application A-Trap Trace)
AB	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(A-Trap Break)
AT	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(A-Trap Trace)
AH	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(A-Trap Heap Zone Check)
AR	[*trap1* [*trap2* [*addr1* [*addr2* [*D1* [*D2*]]]]]]	(A-Trap Record)
AS	*address1* [*address2*]	(A-Trap Spy)
AX		(A-Trap Clear)

Heap zone commands

HX [*address*]	(Heap Exchange)
HC	(Heap Check)
HS [*trap1 trap2*]	(Heap Scramble)
HD [*mask*]	(Heap Dump)
HT [*mask*]	(Heap Total)
SC	(Stack Crawl)

Disassembler commands

SX	(Symbol Exchange)
SD [*address*]	(Symbol Dump)
DH *number*	(Disassemble Hex)
ID [*address*]	(Instruction Disassemble)
IL [*address* [*number*]]	(Instruction List)
F *address count data* [*mask*]	(Find)
WH *expr*	(Where)

Chapter 9

Command Reference

This chapter is a command dictionary that describes each of the Macintosh Workshop commands. Pay particular attention to the "Command Prototype" section, which describes the basic behavior of all commands.

	Command Prototype	xx
AddMenu	Add menu item	xx
Adjust	Adjust lines	xx
Alert	Display an alert box	xx
Alias	Define or write command aliases	xx
Align	Align text to left margin	xx
Asm	68xxx Macro Assembler	xx
Beep	Generate tones	xx
Begin...End	Group commands	xx
Break	Break from For or Loop	xx
C	C Compiler	xx
Canon	Canonical spelling tool	xx
Catenate	Concatenate files	xx
Clear	Clear the selection	xx
Close	Close a window	xx
Compare	Compare text files	xx
Confirm	Display confirmation dialog	xx
Continue	Continue with next iteration of For or Loop	xx
Copy	Copy selection to Clipboard	xx
Count	Count lines and characters	xx
Cut	Copy selection to Clipboard and delete it	xx
CvtObj	Convert Lisa Workshop object files to MPW object files	xx
Date	Write the date and time	xx
Delete	Delete files and directories	xx
DeleteMenu	Delete user-defined menus and items	xx
DeRez	Resource Decompiler	xx
Directory	Set or write the default directory	xx
DumpCode	Write formatted code resources	xx
DumpObj	Write formatted object file	xx
Duplicate	Duplicate files and directories	xx

Echo	Echo parameters xx
Eject	Eject volumes xx
Entab	Convert runs of spaces to tabs xx
Equal	Compare files and directories xx
Erase	Initialize volumes xx
Evaluate	Evaluate an expression xx
Execute	Execute a command file in the current scope xx
Exit	Exit from command file xx
Export	Make variables available to programs xx
FileDiv	Divide a file into several smaller files xx
Files	List files and directories xx
Find	Find and select a text pattern xx
Font	Set font characteristics xx
For...	Repeat commands once per parameter xx
Help	Display summary information xx
If...	Conditional command execution xx
Lib	Combine object files into a library file xx
Link	Link an application, tool, or resource xx
Loop...End	Repeat command list until Break xx
Make	Build up-to-date version of a program xx
MDSCvt	Convert MDS Assembler source xx
Mount	Mount volumes xx
Move	Move files and directories xx
New	Open a new window xx
NewFolder	Create a directory xx
Open	Open a window xx
Parameters	Write parameters xx
Pascal	Pascal Compiler xx
PasMat	Pascal program formatter ("pretty-printer") xx
PasRef	Pascal cross-referencer xx

Paste	Replace selection with contents of the Clipboard xx
Print	Print text files xx
Rename	Rename files and directories xx
Replace	Replace the selection xx
Request	Request text from a dialog xx
Rez	Resource Compiler xx
RezDet	The resource detective xx
Save	Save windows xx
Search	Search files for a pattern xx
Set	Define or write Shell variables xx
SetFile	Set file attributes xx
Shift	Renumber command-file parameters xx
Tab	Set a window's tab value xx
Target	Make a window the target window xx
TLACvt	Convert Lisa TLA Assembler source xx
Unalias	Remove aliases xx
Unmount	Unmount volumes xx
Unset	Remove Shell variables xx
Volumes	List mounted volumes xx
Windows	List windows xx

Command prototype

The following command prototype illustrates the conventions that we've used to describe MPW commands. Most commands behave roughly as specified below.

Syntax

Command [*option*...] [*file*...]

Note: Filenames, command names, and options are not sensitive to case. The syntax notation itself is described in the Preface to this manual.

Description

The first word of the command is the filename of the program to execute, or the name of a predefined command. The subsequent words are passed as additional parameters to the command (or recognized by the Shell in the case of I/O redirection).

Most commands recognize two distinct types of parameters: options and filenames. Options begin with a minus sign (-) to distinguish them from filenames. Although the syntax descriptions list the options first, options and files may appear in any order. All of the options apply to the processing of all of the files, regardless of the ordering of options and files.

For commands that read and write text files, you may specify a file, a window, or a selection within a window, as follows:

name	Named window or file.
§	The selection in the target window. (The target window is the second window from the top.)
name.§	The selection in the named window.

Input

Standard input is often processed if no filenames are specified.

Note: If a program is reading from standard input, you can press Command-Enter (or Command-Shift-Return) to indicate EOF and terminate input. (See "Terminating Input With Command-Enter" in Chapter 3).

Output

Text processors usually write their output to standard output. The Assembler writes listings to standard output. The Linker writes location maps to standard output.

Diagnostics

Errors and warnings are written to diagnostic output. If no errors or warnings are detected, most commands don't write anything to diagnostic output. Assembler and Compiler error messages have the format

```
### message
File "filename" ; Line linenumber
```

This format makes it possible to select and execute the text after "###", because the names "File" and "Line" have been defined as Shell commands— "File" is defined in the Startup file as an alias for the Target command, and "Line" is a short command file that finds a line number.

Several tools write progress and summary information to diagnostic output if you specify the **-p** option.

Status

Status values are returned in the {Status} variable. A value of 0 indicates that no errors occurred; anything else usually indicates an error. Typical values are

0	Command succeeded
1	Incorrect options or parameters
2	Command failed; invalid input

Options

Options specify some variation from the default command behavior. Options begin with a minus sign (-) to distinguish them from files and other parameters.

Options form single words in the command language. Some options require additional parameters, which are separated from the option name with a blank. (An option's parameters also form a single word in the command language.) If more than one option parameter is required, the usual separators between them are commas and equal signs—for example,

```
Asm  -define &debug='on'  -pagesize 84,110  ...
```

For those options that do have additional parameters, the option parameters are never optional.

Options may appear in any order. *All* options are collected prior to processing files.

See also

"Structure of a Command" in Chapter 3

AddMenu — add menu item

Syntax

AddMenu [*menuName* [*itemName* [*command*...]]]

Description

Associates a list of commands with the menu item *itemName*, in the menu *menuName*. If the menu *menuName* already exists, the new item is appended to the bottom of that menu. If the menu *menuName* doesn't already exist, a new menu is appended to the menu bar and the new item is appended to that menu. When the new menu item is selected, its associated command list is executed just as though the command text had been selected and executed in the active window.

Note: The command text that you specify for an AddMenu item is executed twice—once when you execute the AddMenu command itself, and again whenever you subsequently select the new menu item. This means that you must be careful to quote items so that they are processed at the proper time. See the "Examples" section below.

You can also use AddMenu to display information for existing user-defined menus, by omitting parameters:

- If *command* is not specified, the command list associated with *itemName* is written to standard output.
- If *itemName* and *command* are both omitted, a list of all user-defined items for *menuName* is written to standard output.
- If no parameters are specified, a list of all user-defined items is written to standard output.

(This output is in the form of AddMenu commands.)

You can define keyboard equivalents, character styles, and other features for your new menu commands—*itemName* can contain any of the metacharacters that are used with the AppendMenu( ) procedure documented in the "Menu Manager" chapter of *Inside Macintosh*:

/*char*	Assign the keyboard equivalent Command-*char*.
!*char*	Place *char* to the left of the menu item.
^*n*	Item has an icon, where *n* is the icon number (see *Inside Macintosh*).
(	Item is disabled (dimmed).
<*style*	Item has a special character style: *style* can be any of the following:

B	bold

I	italic
U	underline
O	outline
S	shadow

Be sure to quote menu items containing these special characters. (See the "Examples" section below.)

Note: Semicolons (;) ***cannot*** be used within an ***itemName.***

Menu items can't be appended to the Windows, Format, or Apple menus.

Input None.

Output If any of the optional parameters is omitted, a list of user-defined menu items and their associated commands is written to standard output.

Diagnostics Errors and warnings are written to diagnostic output.

Status AddMenu returns the following status values:

0	No errors
1	Syntax error
2	An item can't be redefined
3	System error

Examples

```
AddMenu
```

Lists all user-defined menu items.

```
AddMenu Extras "TimeStamp/P"  'Echo `Date`'
```

Adds an "Extras" menu with a "TimeStamp" item, which writes the current time and date to the active window. This item has the Command-key equivalent Command-P.

```
AddMenu File 'Format<B' 'Erase 1'
```

Adds a "Format" item to the File menu (see the Erase command), and makes the item bold.

```
AddMenu Find Top 'Find • "{Active}"'
```

Adds the menu item "Top" to the Find menu, and defines it as the Find command enclosed in single quotes—this command places the insertion point at the beginning of the active window.

Note: The following attempt to do the same thing *will not work.*

```
AddMenu Find Top "Find • {Active}"
```

This command 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.*) In the first (correct) example, the single quotes defeat variable expansion when the AddMenu command is executed; they are then stripped off before the item is actually added. The double quotes remain, in case the pathname of the active window happens to contain any special characters.

You may want to add some or all of the following commands to your UserStartup file:

```
AddMenu Find '(-'                     ''
AddMenu Find 'Top/6'                  'Find • "{Active}"'
AddMenu Find 'Bottom/5'               'Find ∞ "{Active}"'
AddMenu Find 'Clear to Bottom/8' 'Clear §:∞ "{Active}"'
```

These commands create several new items in the Find menu. The first is a disabled separator that creates a new section at the bottom of the menu. The Top and Bottom items position the insertion point at the top and bottom of the active window. Clear to Bottom clears everything from the beginning of the current selection (or insertion point) to the end of the active window. All three menu items have Command-key equivalents.

See also

DeleteMenu command

"Quoting Special Characters," "How Commands Are Interpreted," and "Defining your own Menu Commands" in Chapter 3.

"Creating a Menu in Your Program" in the "Menu Manager" chapter of *Inside Macintosh*

Adjust — adjust lines

Syntax

Adjust [-c *count*] [-l *spaces*] *selection* [*window*]

Description

Finds and selects the given selection, and shifts all lines within the selection to the right by one tab, without changing the indentation.

If a count is specified, *count* instances of selection are affected. The **-l** option lets you move lines by any number of spaces to the left or right.

If you specify the *window* parameter, the command operates on *window*. It's an error to specify a window that doesn't exist. If no window is specified, the command operates on the target window (the second window from the top).

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Adjust returns the following status values:

0	At least one instance of the selection was found
1	Syntax error
2	Anything else

Options

-c *count*	Repeat the select-and-adjust operation *count* times.
-l *spaces*	Every line within the selection will be shifted *spaces* spaces to the right. You can shift a selection left by specifying a negative value for *spaces*.

Examples

```
Adjust -l 4 §
```

Shifts the lines containing the target selection to the right by four spaces.

```
Adjust -l -8 /if/Δ:Δ/else/
```

Selects everything after the next "if" and before the following "else", and shifts all lines within the selection to the left by eight spaces.

See also Align command
"Selections" in Chapter 4

Alert — display an alert box

Syntax Alert *message*

Description Displays an alert box containing the prompt *message*. The alert is displayed until its OK button is clicked. If the message consists of more than one word, or contains any special characters, you'll need to quote it, as explained in Chapter 3.

Input None.

Output None.

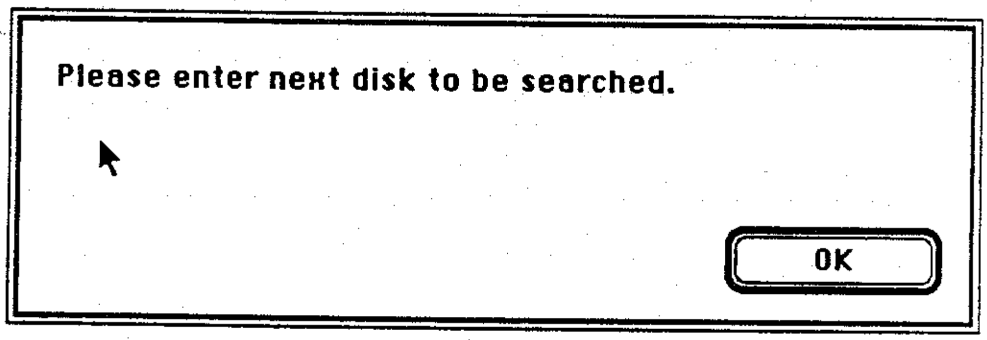
Diagnostics None.

Status The Alert command normally returns the value 0. The value 1 is returned if there were any syntax errors.

Examples

```
Alert "Please enter next disk to be searched."
```

Displays the following alert box, and waits for the user to click "OK" before returning.



Please enter next disk to be searched.

See also Confirm and Request commands

Alias — define or write command aliases

Syntax

Alias [*name* [*word*...]]

Description

Name becomes an alias for the list of words. Subsequently, when *name* is used as a command name, *word*... will be substituted in its place.

If only *name* is specified, any alias definition associated with *name* is written to standard output. If *name* and *word* are both omitted, a list of all aliases and their values is written to standard output. (This output is in the form of Alias commands.)

Aliases are local to the command file in which they are defined. An initial list of aliases is inherited from the enclosing command file. Inherited aliases may be overridden locally. You can make an alias definition available to all command files by placing the definition in the UserStartup file.

You can remove aliases with the Unalias command.

Input

None.

Output

When parameters are omitted, the Alias command writes aliases and their values to standard output.

Diagnostics

None.

Status

A status value of 0 is always returned.

Examples

```
Alias CD Directory
```

Creates an alias "CD" for the Directory command.

```
Alias Top 'Find •'
```

Creates an alias "Top" for the command "Find •" (which places the insertion point at the beginning of a window). The command takes an optional window parameter, and by default acts on the target window. The Top command could now be used as follows:

```
Top            # find top of target window
Top Sample.a   # find top of window Sample.a
               # (equivalent to "Find • Sample.a")
```

The following example redefines an existing command:

```
Alias Save SaveMany
```

The built-in Save command does not allow you to save a list of windows. To override the built-in Save command with your own version of the command, you could alias Save to the command file SaveMany, which might contain the following:

```
#### SaveMany - Save a list of windows ####
#
#    SaveMany [window...]
Unalias    # It's very important to Unalias Save!
Set Exit 0
For window in {"Parameters"}
     Save "{window}"
End
```

The Unalias command must be included—it removes the alias for Save, preventing infinite recursion when Save is used later in the command file. To make this multi-window save a permanent feature on your system, you could put the Alias command in your UserStartup file, and put the SaveMany command file in the Tools directory.

See also

Unalias command

"Command Aliases" in Chapter 3

Align — align text to left margin

Syntax

Align [-c *count*] *selection* [*window*]

Description

All lines within each instance of the selection are positioned to the same distance from the left margin as the first line in the selection.

If you specify the *window* parameter, the Align command will act on *window*. It's an error to specify a window that doesn't exist. If no window is specified, the command operates on the target window (the second window from the top).

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Align returns the following status values:

0	At least one instance of the selection was found
1	Syntax error
2	Any other error

Options

-c *count*	Repeat the select-and-align operation *count* times.

Examples

```
Align §
```

Same as the Align menu item; that is, aligns all lines in the default selection with the first line of the selection.

```
Align /Begin/:/End/
```

Selects everything from the next "Begin" through the following "End", and aligns all lines within the selection to the same margin position as the line that contains the "Begin".

See also

Adjust command

"Selections" in Chapter 4

Asm — 68xxx Macro Assembler

Syntax

Asm [*option* …] [*file* …]

Description

Assembles the specified assembly-language source files. One or more filenames may be specified. If no filenames are specified, standard input is assembled and the file "a.o" is created. By convention, assembly-language source file names end in the suffix ".a". Each file is assembled separately— assembling file *name*.a creates object file *name*.a.o. The object file name can be changed with the **-o** option.

See the *MPW Assembler Reference* manual for details of the assembly language.

Input

If no filenames are specified, standard input is assembled. (You can terminate input by typing Command-Enter.)

Output

If either the **-l** or **-s** option is specified, an assembler listing is generated. If standard input is used for the source file, the listing is written to standard output. If the input is taken from file *name*.a, the listing is written to *name*.a.lst. The listing file name can be changed with the **-lo** option.

Diagnostics

Errors and warnings are written to diagnostic output. If the **-p** option is specified, progress and summary information is also written to diagnostic output.

Status

The following status values are returned to the Shell:

0	No errors detected in any of the files assembled
1	Parameter or option errors
2	Errors detected
3	Execution terminated

Options

Except for the **-case on** option, options may appear in any order.

-addrsize *size*	Set address displays in the listing to *size* digits (values 4 to 8 are allowed). The default is 5 digits.

-blksize *blocks* Set the Assembler's text file I/O buffer size to *blocks**512 bytes. Values 6 to 62 are allowed. Odd values are made even by reducing the value by 1. The default value is 16 (8192 bytes) if the Assembler determines it has the memory space for the I/O buffers, and 6 (3072 bytes) otherwise. This option permits optimization of I/O performance (transfer rate for text file input, load/dump files, and listing output) as a function of the disk device being used. Note that increasing the blocks value reduces the amount of memory available for other Assembler structures (such as symbol tables).

-case on Distinguish between upper- and lowercase letters in non-macro names (same as CASE ON). (Case is always ignored in macro names.) If you intend to preserve the case of names declared by the **-define** option, then the **-case on** option must *precede* the **-define** option(s) in the command line.

-case obj[ect] Preserve the case of module, EXPORT, IMPORT, and ENTRY names ***only in the generated object file***. In all other respects, case is ignored within the assembly, and the behavior is the same as the preset CASE OFF situation.

-case off Ignore the case of letters. All identifiers are case insensitive. This is the preset mode of the Assembler, but it may be used in the command line to reverse the effect of one of the other **-case** modes.

-c[heck] Syntax check only. No object file is generated.

-d[efine] ***name*[=*value*]** [,***name*[=*value*]**]...

Define the name as having the specified value. The value is a decimal integer. If ***value*** is omitted, a value of 1 is assumed. This option is equivalent to placing the directive

name `EQU` ***value***

at the beginning of your source file. Note that in order to test whether or not the name is defined, the `&Type` function should be used. You can define more than one name by specifying multiple -d options or multiple ***name*[=*value*]** parameters separated by commas. For example,

```
Asm -d debug1,&debug='on'  ...
```

-d[efine] ***&name*[=[*value*]] [,*&name*[=[*value*]]]...**

Define the macro ***name*** as having the specified value. The value is a decimal integer or a string constant. If the "***=value***" is omitted, the decimal value 1 is assumed. If only the ***value*** is omitted, the null string is assumed. **-define** is equivalent to declaring the name as a global arithmetic symbol (GBLA for an integer value) or global character macro symbol (GBLC for a string value) and placing one of the following directives at the beginning of the source file:

```
&name SETA value
&name SETC value
```

Note that in order to test whether the name is defined, the `&Type` function should be used. You can define more than one macro name by specifying multiple **-d** options or multiple *&name*[=*value*] parameters separated by commas.

-e[rrlog] ***filename***

Write all errors and warnings to the error log file with the specified filename (same as ERRLOG '***filename***').

-f Suppress page ejects (same as PRINT NOPAGE).

-font ***fontname*[,*fontsize*]**

Set the listing font to ***fontname*** (for example, Courier), and the size to ***fontsize***. This option is meaningful only if the **-s** or the **-l** option is used. The default listing font is Monaco 7. Note that listings will be formatted correctly only if a monospaced font is used.

-h Suppress page headers (same as PRINT NOHDR).

-i ***pathname*** **[,*pathname*]...**

Search for include and load files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is as follows:

1. The include or load filename is used as specified. If a ***full pathname*** is given, then no other searching is applied.

   If the file wasn't found, and the pathname used to specify the file was a ***partial pathname*** (no colons in the name or a leading colon), then the following directories are searched.
2. The directory containing the current input file.
3. The directories specified in **-i** options, in the order listed.
4. The directories specified in the Shell variable {AIncludes}.

-l Generate full listing. If file *name*.a is assembled, the listing is written to *name*.a.lst.

-lo *listingname* Pathname for the listing file and directory for the listing scratch file. If *listingname* ends with a colon (:), it indicates a directory for the listing file, whose name is then formed by the normal rules (that is, *inputFilename*.a.lst). If *listingname* does not end with a colon, the listing file is written to the file *listingname*. In this case, listings for multiple source files are appended to the listing file. In either case, the directory implied by the listing name is used for the Assembler's listing scratch file. The **-lo** option is only meaningful if the **-s** or the **-l** option is used.

-o *objname* Pathname for the generated object file. If *objname* ends with a colon (:), it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputFilename*.o). If *objname* does not end with a colon, the object file is written to the file *objname*. (In this case, only one source file should be specified to the Assembler.)

-pagesize *l* [,*w*] Set the listing page size. (This option is only meaningful if the **-s** or **-l** option is specified.) The *l* and *w* parameters are integers: *l* is the page length (default = 75) and *w* is the page width (default = 126). (These settings assume that Monaco 7 is being used with the MPW Print command to the LaserWriter.)

-print *mode* [,*mode*]...

Set a print option mode. *Mode* may be any one of the following PRINT directive options:

[NO]GEN	macro expansions
[NO]PAGE	page ejects
[NO]WARN	warnings
[NO]MCALL	macro calls
[NO]OBJ	object code
[NO]DATA	data
[NO]MDIR	macro directives
[NO]HDR	page headings
[NO]LITS	literals
[NO]STAT	progress information
[NO]SYM	symbol table display

See the *MPW Assembler Reference* manual for a discussion of these PRINT settings. You can specify more then one print option by specifying multiple **-print** options or multiple *mode* parameters separated by commas. For example,

```
Asm -print nowarn,noobj,nopage  ...
```

Note that single-letter options are provided for some of the settings: **-f** (NOPAGE), **-h** (NOHDR), **-p** (STAT), and **-w** (NOWARN).

-p	Write assembly progress information (module names, included, loads, and dumps) and summary information (number of errors, warnings, and compilation time) to the diagnostic output file. (This option is the same as PRINT STAT.)
-s	Set PRINT NOOBJ to generate a shortened form of the listing file. If the **-l** option is also specified, the rightmost option takes precedence.
-t	Display the assembly time and the number of lines to the diagnostic file even if progress information (**-p**) is not being displayed.
-w	Suppress warning messages (same as PRINT NOWARN).
-wb	Suppress branch warning messages only.

Examples

```
Asm -w -l Sample.a Memory.a -d Debug
```

Assembles Sample.a and Memory.a, producing object files Sample.a.o and Memory.a.o. Suppresses warnings and define the name "Debug" as having the value 1. Two listing files are generated: Sample.a.lst and Memory.a.lst. (Sample.a dud Memory.a are located in the AExamples directory.)

See also

MPW Assembler Reference

Beep — generate tones

Syntax

Beep [*note* [,*duration* [,*level*]]]...

Description

For each parameter, Beep produces the given note for the specified duration and sound level on the Macintosh speaker. If no parameters are given, a simple beep is produced.

Note is one of the following:

- A number indicating the count field for the square wave generator, as described in the Summary of the "Sound Driver" chapter of *Inside Macintosh*
- A string in the following format:

  [*n*] *letter*[# | b]

  n is an optional number between -3 and 3 indicating the octaves below or above middle C, followed by a letter indicating the note (A–G) and an optional sharp (#) or flat (b) sign.

The optional *duration* is given in sixtieths of a second. The default duration is 15 (one-quarter second).

The optional sound *level* is given as a number from 0 to 255. The default level is 128.

Input

None.

Output

None.

Diagnostics

None.

Status

A status value of 0 is always returned.

Examples

```
Beep
```

Produce a simple beep on the speaker.

```
Beep 2C,20 '2C#,40' 2D,60
```

Play the 3 notes specified: C , C sharp, and D, all two octaves above middle C, for one-third, two-thirds, and one full second respectively. Notice that the second parameter must be quoted; otherwise the sharp character (#) would indicate a comment.

Begin...End — group commands

Syntax

```
Begin
    command...
End
```

Description

Groups commands for pipe specifications, conditional execution, and input/output specifications. Carriage returns must appear at the end of each line as shown above, or be replaced with semicolons (;). If the pipe symbol (|), conditional execution operators (&& and ||), or input/output specifications (<, >, >>, ≥, ≥≥) are used, the operator must appear after the End command, and applies to all of the enclosed commands.

Note: Begin and End behave like left and right parentheses. Once the Begin command has been executed, the Shell will not execute any of the subsequent commands until it encounters the End command, so that input/output specifications can be processed.

Input

None.

Output

None.

Diagnostics

None.

Status

The status value of the last command executed is returned. (If no commands appear between Begin and End, 0 is returned.)

Examples

The following commands save the current variables, exports, aliases, and menus in file SavedState.

```
Begin
   Set
   Export
   Alias
   AddMenu
End > SavedState
```

Notice that the output specification following "End" applies to all of the commands within the Begin...End control command. This command is identical to the following:

```
(Set; Export; Alias; AddMenu) > SavedState
```

The commands Set, Export, Alias, and AddMenu write their output in the form of commands; these commands can be executed to redefine variables, exports, aliases, and menus. Therefore, after executing the above commands, the command

```
SavedState
```

will restore all of these definitions.

Note: This technique is used in the Suspend command file to save state information. (You might want to take a look at Suspend, which also saves the list of open windows and the current directory.) The Resume file runs the file that Suspend creates, restoring the various definitions, reopening the windows, and resetting the current directory.

Break — break from For or Loop

Syntax

Break [If *expression*]

Description

If *expression* is nonzero, Break terminates execution of the immediately enclosing For or Loop command. (Null strings are considered zero.) If the "If *expression*" is omitted, the break is unconditional. (For a definition of *expression*, see the Evaluate command in this chapter.)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	No errors detected
1	Break is found outside a For…End or Loop…End, or the parameters to Break are incorrect

Examples

```
Set Exit 0
For file in Startup UserStartup Suspend Resume Quit
     EnTab "{file}" > temp
     Break If {Status} != 0
     Rename -y temp "{file}"
     Print -h "{file}"
     Echo "{file}"
End
```

This For loop entabs and prints each of the special MPW command files; the Break command terminates the loop if a nonzero status value is returned. (See the For command for an explanation of this example.)

See also

For, Loop, and If commands

Evaluate command (for a description of expressions)

"Structured Commands" in Chapter 3

C — C Compiler

Syntax

C [*option* ...] [*file*]

Description

Compiles the specified C source file. Compiling file *Name*.c creates object file *Name*.c.o. (By convention, C source file names end in a ".c" suffix.) If no filenames are specified, standard input is compiled and the object file "c.o" is created.

See the manual *MPW C Reference* for details of the C language definition.

Input

If no filenames are specified, standard input is compiled. You can terminate input by pressing Command-Enter.

Output

If you specify the **-e** option, preprocessor output is written to standard output, and no object file is produced.

Diagnostics

Errors and warnings are written to diagnostic output. If the **-p** option is specified, progress and summary information is also written to the diagnostic output.

Status

The following status values are returned:

0	Successful completion
1	Errors occurred

Options

-c Include comments with the preprocessor output. (By default, comments are not written to the preprocessor output.)

-d *name* Define ***name*** to the preprocessor with the value 1. This is the same as writing

`#define` *name* `1`

at the beginning of the source file. (The **-d** option does not override `#define` statements in the source file.)

-d *name=string* Define ***name*** to the preprocessor with the value *string*. This is the same as writing

`#define` *name string*

at the beginning of the source file.

-e Do not compile the program. Instead, write the output of the preprocessor to standard output. This option is useful for debugging preprocessor macros.

-g Generate stack frame pointers in A6 (that is, LINK A6,x ... UNLK A6) for all functions. Insert the procedure name into the object code that follows the procedure's RTS instruction. Use this option if you plan to debug the program with MacsBug.

-ga Generate stack frame pointers in A6 (that is, LINK A6,x ... UNLK A6) for all functions.

-i *pathname* [,*pathname*]...

Search for include files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is as follows:

1. The include file name is used as specified. If a *full pathname* is given, then no other searching is applied.

   If the file wasn't found, and the pathname used to specify the file was a *partial pathname* (no colons in the name or a leading colon), then the following directories are searched.
2. The directory containing the current input file.
3. The directories specified in **-i** options, in the order listed.
4. The directories specified in the Shell variable {CIncludes}.

-o *objname* Pathname for the generated object file. If *objname* ends with a colon (:), it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputFilename*.o). If *objname* does not end with a colon, the object file is written to the file *objname*.

-p Write progress information (include file names, function names, and sizes) and summary information (number of errors and warnings, code size, global data size, compilation time, and compilation memory requirements) to diagnostic output.

-q

Optimize the code for speed, even if it's necessary to make the object code larger. By default, the Compiler performs optimizations that make the code both smaller and quicker—the -q option will perform further optimizations that may make the code faster, but also larger. The **-q** option should be specified only for those parts of the program that are executed frequently—it's counterproductive to specify **-q** on code that's rarely executed.

-q2

Allow the optimizer to assume that memory locations do not change except by explicit stores—that is, the optimizer is guaranteed (1) that no memory locations are I/O registers that can be changed by external hardware, and (2) that no memory locations are shared with other processes that can change them asynchronously with respect to the current process. This option must be used with extreme caution in device drivers, operating systems, and shared-memory environments, and when interrupts are present.

-s ***name***

Name the object code segment. (The default segment name is "Main".) Because a segment may not exceed 32K bytes, large programs require multiple segments with different names. This option is overridden if the following statement appears in the source code:

```
#define __SEG__ name
```

-u ***name***

Undefine the predefined preprocessor symbol *name*. This is the same as writing

```
#undef name
```

at the beginning of the source file.

-w

Suppress Compiler warning messages. (By default, warnings are written to diagnostic output.)

-x6

Use MOVE #0,x instructions rather than CLR x instructions for nonstack addresses. This option may be useful when writing device drivers.

-x55

Make bit fields of types int, short, and char be signed. (The default is for all fields to be unsigned.)

-z6	Always allocate 32 bits for enumerated data types, to maintain compatibility with Standard C. The default is to allocate 8, 16, or 32 bits. **Caution:** This option is not compatible with the Macintosh interface libraries.
-z84	Enable language anachronisms. Warning messages are provided when anachronisms are encountered, and the constructs are compiled. (See *MPW C Reference* for information.)

Examples

```
C -p Sample.c
```

Compile Sample.c, producing the object file Sample.c.o. Write progress information to diagnostic output. (Sample.c is found in the CExamples folder.)

Limitations

1 MB of RAM is recommended; on a Macintosh 512K, even small C programs may not compile.

Availability

The C Compiler is available as part of a separate Apple product, Macintosh Programmer's Workshop C.

See also

MPW C Reference

Canon — canonical spelling tool

Syntax

Canon [-s] [-a] [-c *n*] *dictionaryFile* [*inputFile* …]

Description

Canon copies the specified files to standard output, replacing identifiers with the canonical spellings given in *dictionaryFile*. If no files are specified, standard input is processed.

DictionaryFile is a text file that specifies the identifiers to be replaced and their new (or canonical) spellings. Identifiers are defined as a letter followed by any number of letters or digits (underscore (_) is also considered a letter). Each line in the dictionary contains either a pair of identifiers or a single identifier:

- If *two identifiers* appear, the first is the identifier to replace, and the second is its canonical spelling. For example, the dictionary entry

  ```
  NIL  NULL    # change NIL to NULL
  ```

  changes each occurrence of NIL to NULL.

- A *single identifier* specifies both the identifier to match and its canonical spelling—this feature is useful because the matching may be case insensitive or restricted to a fixed number of characters. (See the "Options" section below.) For example, the dictionary entry

  ```
  true
  ```

  changes all occurrences of "TRUE", "True", "tRUE", and so on to "true".

You can specify a left context for the first identifier on each line of the dictionary by preceding it with a sequence of non-identifier characters. Replacement will then occur only if the left context in the input file exactly matches the left context in the dictionary. For example, if C structure component upperLeft should be replaced with topLeft, the dictionary might include the following:

```
.upperLeft  topLeft
      ->upperLeft  topLeft
```

You can include comments in the dictionary file by using the # symbol—everything from the # to the end of the line is ignored.

Note: The file Canon.Dict is a sample dictionary file that's included with MPW. (See the "Examples" section below.)

Input

Standard input is read if no files are specified.

Output

The specified files are written to standard output with the identifiers replaced. (Words in comment sections are also replaced.)

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	All files processed successfully
1	Error in command line
2	Other errors

Options

-s	Use case-sensitive matching. (Pattern matching is normally case insensitive.)
-a	Causes the characters $, %, and @ to be considered letters (for defining identifiers). This option is useful when processing assembly-language source.
-c *n*	Take only the first *n* characters as significant. (Normally all characters in identifiers are significant.)

Examples

The file Canon.Dict, in the Tools folder, contains a list of all of the identifiers used in the Standard C library and the *Inside Macintosh* C interfaces. This list was made from the Library Index in the *MPW C Reference* manual. The entries in Canon.Dict look like the following:

```
abbrevDate
ABCallType
abortErr
ABProtoType
abs
acos
activateEvt
...
```

The following command copies the file Source.c to the file Temp; identifiers whose first eight characters match a dictionary entry are replaced with that entry.

```
Canon -c 8 "{MPW}"Tools:Canon.Dict Source.c > Temp
```

The `-c 8` option is useful when porting source from other systems where only eight characters are significant.

Note: The list of Pascal identifiers used in the *Inside Macintosh* interface is almost identical to the list used in C. The dictionary Canon.Dict can also be used to port Pascal programs from other systems, as long as you don't mind using the canonical capitalizations for the various Standard C library identifiers.

Limitations The maximum line length in the dictionary file is 256 characters. Longer lines are considered an error.

Catenate — concatenate files

Syntax

Catenate [*file* ...]

Description

Catenate reads the data fork of each file in sequence and writes it to standard output. If no input file is given, Catenate reads from standard input. None of the input files may be the same as the output file.

Input

Standard input is processed if no filenames are specified.

Output

All files are written to standard output.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	All files were processed successfully
1	One or more files were not found
2	An error occurred in reading or writing

Examples

```
Catenate Makefile.a
```

Writes Makefile.a to the active window, immediately following the command.

```
Catenate File1 File2 > CombinedFile
```

Concatenates the first two files and places the result in the third. If CombinedFile doesn't exist, it will be created; if it exists, it will be overwritten.

```
Set selection `Catenate §`
```

Capture the selection from the target window in the Shell variable {selection}.

```
Catenate >> {Worksheet}
```

Append all subsequently entered text to the Worksheet window (until end-of-file is indicated by pressing Command-Enter).

Warnings

Beware of commands such as

```
Catenate File1 File2 > File1
```

This command will cause the original data in File1 to be lost. To append one file to another, use the form

```
Catenate File2 >> File1
```

See also

Duplicate command

"Redirecting Input and Output" in Chapter 3

Clear — clear the selection

Syntax

Clear [-c *count*] *selection* [*window*]

Description

Finds *selection* and deletes its contents. The selection is not copied to the Clipboard. (For a definition of *selection*, see Chapter 4.)

If *window* is specified, the Clear command acts on that window. It's an error to specify a window that doesn't exist. If no window is specified, the command operates on the target window (the second window from the top).

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Clear returns the following status values:

0	At least one instance of *selection* was found
1	Syntax error
2	Anything else

Options

-c *count*	Repeat count—find and delete *count* instances of selection.

Examples

```
Clear §
```

Deletes the current selection. This is like the Clear command in the menu bar, except that the action occurs in the target window rather than the active window.

```
Clear /BEGIN/:/END/
```

Select everything from the next BEGIN through the following END, and delete the selection.

See also

Cut and Replace commands

"Selections" in Chapter 4 (see Appendix B for a summary)

Close — close a window

Syntax

Close [-y | -n] [*window*]

Description

Close *window*. If no window is specified, the target window is closed. If changes to the window haven't been saved, a dialog box requests confirmation of the Close. In command files, you can use the **-y** or **-n** options to avoid this interaction.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Close returns the following status values:

0	No errors
1	Syntax error
2	Any other error

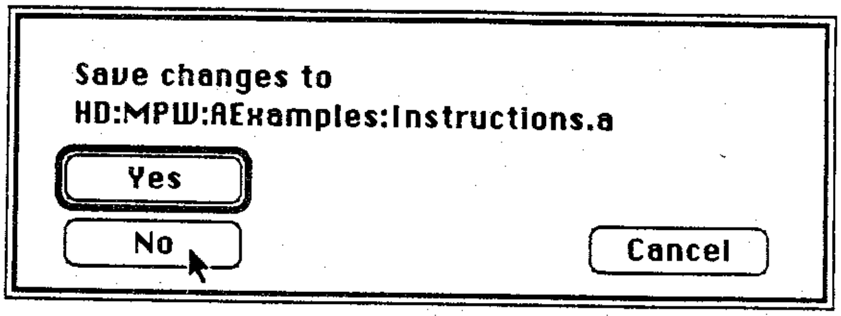
Options

-y	Answer "Yes" to the confirmation dialog, causing the contents of *window* to be saved before closing it.
-n	Answer "No" to the confirmation dialog, causing *window* to be closed without saving any changes.

Examples

```
Close Instructions.a
```

Close and save the window titled Instructions.a. If any changes had been made to the file, the following dialog would appear:



You can bypass this dialog with the **-y** or **-n** options.

Compare — compare text files

Syntax

Compare [*option* ...] *file1* [*file2*]

Description

Compares the lines of two text files and writes their differences to standard output. Options are provided to compare a specific column range in each file (**-c**), to ignore blanks (**-b**), and to ignore case (**-l**).

Both files are read and compared line-for-line. As soon as a mismatch is found, the two mismatched lines are stored in two stacks, one for each file. Lines are then read alternately (starting from the next input line in *file2*) until a match is found to put the files back in synchronization. If such a match is found, Compare writes the mismatched lines to standard output.

Files are considered resynchronized when a certain number of lines in the two stacks exactly match. By default, the number of lines, called the *grouping factor*, is defined by the formula

$$G = \text{Trunc}((2.0 * \text{Log}_{10}(M)) + 2.0)$$

where *G* is the grouping factor and *M* is the number of lines saved in each stack so far. This definition requires more lines to be the same after larger mismatches. Using this formula, the following table shows the grouping factor, *G*, as a function of the number of mismatched lines:

M: *Number of mismatched lines*			G: *Grouping factor*
1	to	3	2
4	to	9	3
10	to	31	4
32	to	99	5
100	to	315	6
316	to	999	7
1000	to	3161	8
3162	to	9999	9

With the default dynamic grouping, the **-g** option sets the lower limit for *G* (which must be at least 2, since the formula is always applied). The **-s** option lets you fix *G* as a static constant. A static *G* may be desirable under some circumstances, but may also resynchronize the files at undesirable points, especially if *G* is too small. It's recommended that you use the default (dynamic *G*) first; if the results aren't satisfactory, try the static *G* option.

With either option, there's a limit on the depth of the stacks; that is, to how far out of synch the two files can get before they're no longer worth comparing. For a dynamic *G*, the limit on the number of mismatched lines is 1000, but you can choose a lower limit with the **-d** option. For the static *G* option, typical values for *G* are 1 to 5, and the stack depth should be between about 10 and 50 (the default limit is 25).

Input

The *file1* and *file2* parameters specify the two files to be compared. If *file2* is omitted, *file1* is compared to standard input.

Output

Mismatched lines and descriptive messages are written to standard output. With the **-h** option, a portion of each file's output lines are displayed side by side; otherwise, the first stack's lines are displayed before the second stack's. In either case, lines are shown with their line numbers.

The following messages appear when showing mismatches:

`Nonmatching lines`
    ...both stacks are displayed...

`Extra lines in 1st before <`*line*`> in 2nd`
    ...lines in file1's stack are displayed...

`Extra lines in 2nd before <`*line*`> in 1st`
    ...lines in file2's stack are displayed...

`Extra lines in 1st file`
    ...lines in file1's stack are displayed...

`Extra lines in 2nd file`
    ...lines in file2's stack are displayed...

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

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

If both files are in synchronization, and both reach their end-of-file at the same time, then the following message will appear *if any* mismatches occurred:

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

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

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

Diagnostics

Parameter errors are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Files match
1	Parameter or option error
2	Files don't match
3	Execution terminated

Options

-b Treat several blanks (spaces or tabs) as a single space, and ignore trailing blanks.

-c *col1-col2*[*,col1-col2*]
Compare only the columns *col1* to *col2* of each file. If the second column range is omitted, then the first range applies to both files; otherwise the first range applies to *file1* and the second range applies to *file2*. If *col1* is omitted, 1 is assumed. If *col2* is omitted, 255 is assumed.

Note: To use the **-c** option, tabs must be expanded. The tab setting is determined from the file's tab value. (See also the **-x** option below.)

-d *depth* Sets the maximum stack depth (size) for resynchronization, that is, how far out of synch the files can get before they're no longer worth comparing. *Depth* is an integer value from 1 to 1000. The default is 1000 if dynamic grouping is being used, and 25 for static (**-s**) grouping.

-g *groupingFactor*
Specifies the grouping factor, *G*. For dynamic grouping, **-g** specifies the ***minimum*** grouping factor, that is, the minimum number of lines that must match for the two files to be considered resynchronized. (This value must be at least 2, which is the default.) If the **-s** (static) option is used, **-g** specifies a fixed grouping factor. (Values are from 1 to 1000; the default is 3.)

-h *width* Display mismatches in the horizontal format. Only a portion of each mismatched line is displayed side by side. Width is a number from 70 to 255 that controls the number of characters displayed in each portion by specifying the total display line width.

-l Ignore case differences (convert all lines to lowercase before comparing them). The default is case sensitive.

-n Do not write any messages to standard output if both files match.

-p Write Compare's version information to diagnostic output.

-s Static (fixed) grouping factor (the grouping factor is set with the -g option).

-t Ignore trailing blanks (spaces or tabs). (This is a subset of the **-b** option.)

-x Suppress tab expansion. Normally, except when the **-b** option is used, tabs are expanded into spaces. The tab value is determined from the file's tab setting (a resource); if there is no setting, 4 is used.

Caution: This option can cause stacked lines to be displayed incorrectly if the files contain tabs. Also, the **-c** option should not be used with **-x**, because **-c** depends on the true columns as displayed with tabs expanded.

Note: All comparison criteria that affect the individual lines *before* comparison—column range (**-c**), blanks compression (**-b**), and case conversion (**-l**)—are applied to those lines before they are stacked. Thus when the lines are displayed, they'll be shown in their modified form.

Examples

```
Compare File File.bak > Mismatches
```

Compare File and File.bak, writing the results to the file Mismatches. No options are specified, so dynamic grouping is used, blanks are retained, tabs are expanded into spaces, and matching is case sensitive.

```
Compare  File.old.§  File.new.§
```

Compares the selected portions of the two windows and writes out the results.

Limitations

Compare can handle text files with a maximum line length of 255 characters.

The text files compared should be fewer than 9999 lines long, because the displays are formatted based on four-digit line numbers.

See also

Equal command (Equal is a quicker command that tells you whether files are different, and stops at the byte at which they differ.)

Confirm — display confirmation dialog

Syntax

Confirm [-t] *message*

Description

Displays a confirmation dialog with OK and Cancel buttons and the prompt *message*. There is no output to this command—the result of the dialog is returned in the {Status} variable.

Note: Because Confirm returns a nonzero status value to indicate that No or Cancel was selected, a command file should set the Shell variable {Exit} to zero before executing the Confirm command. (This step is necessary because the Shell aborts command file processing when a nonzero status value is returned and {Exit} is nonzero.)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The Confirm command returns the following status values:

0	The OK (or yes) button was selected
1	Syntax error
4	The Cancel (or no) button was selected
5	The Cancel button was selected in a three-way dialog—see the -t option

Note: In the context of a two-button dialog, Cancel means the same thing as No.

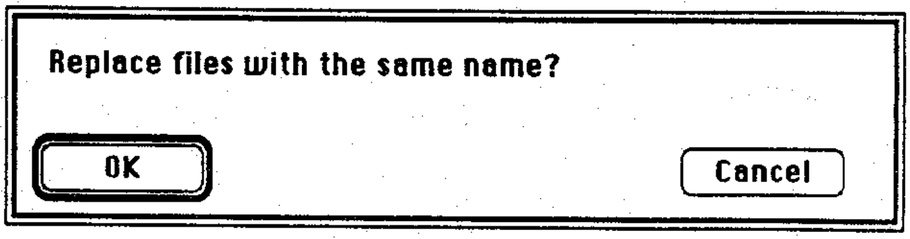
Options

-t	Display a three-way confirmation dialog, which includes Yes, No, and Cancel buttons. In this case, 4 means No and 5 means Cancel.

Examples

```
Set Exit 0
Confirm "Replace files with the same name?"
If {Status} == 0
   Duplicate -y Source: Destination:
End
Set Exit 1
```

The following confirmation dialog will be displayed:



If the user selects the OK button, the Duplicate command will be executed.

The following command file makes use of a three-way confirmation dialog:

```
Set Exit 0
Set list ""
For file In `files -t TEXT`
     Confirm -t "Print file {file}?"
     Set SaveStatus {Status}
     Continue If {SaveStatus} == 4      # No
     Break If {SaveStatus} == 5         # Cancel
     Set list "{list} '{file}'"         # Yes
End
Print {PrintOptions} {list}
Set Exit 1
```

This example prints selected TEXT files in the current directory. For each file, it displays a dialog with three choices (Yes, No, and Cancel). Selecting Yes prints the file. If you select No, the Continue command causes this file to be skipped, but processing continues with the next file in the list. If you select Cancel, the Break command causes the For loop to be terminated, ending the question-and-answer session. The filenames are saved in the variable {list}, and printed following the loop.

See also Alert and Request commands

Continue — continue with next iteration of For or Loop

Syntax

Continue [If *expression*]

Description

If *expression* is nonzero, Continue terminates this iteration of the immediately enclosing For or Loop command, and continues with the next iteration. (Null strings evaluate to zero.) If the "If *expression*" clause is omitted, the Continue is unconditional. If no further iterations are possible, the For or Loop is terminated. (For a definition of *expression*, see the Evaluate command in this chapter.)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Continue returns the following status values:

0	No errors
1	Error in parameters, or Continue not within For...End or Loop...End

Examples

```
Set Exit 0
Set list ""
For file In `files -t TEXT`
     Confirm -t "Print file {file}?"
     Set SaveStatus {Status}
     Continue If {SaveStatus} == 4     # No
     Break If {SaveStatus} == 5        # Cancel
     Set list "{list} '{file}'"        # YesEnd
Print {PrintOptions} {list}
Set Exit 1
```

In this example, the Continue command is executed if the user selects No (status value 4). The Continue causes the current file to be skipped, but processing continues with the next file in the list.

(For a full explanation of this example, refer to the Confirm command on the previous page.)

See also

For, Loop, Break, and If commands

Evaluate command for a description of expressions
"Structured Commands" in Chapter 3

Copy — copy selection to Clipboard

Syntax

Copy [-c *count*] *selection* [*window*]

Description

Finds *selection* in the specified window and copies it to the Clipboard, replacing the previous contents of the Clipboard. If no window is specified, the command operates on the target window (the second window from the top). It's an error to specify a window that doesn't exist.

For a definition of *selection*, see "Selections" in Chapter 4; a summary of the selection syntax is contained in Appendix B.

Note: To copy *files*, use the Duplicate command.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Copy returns the following status values:

0	At least one instance of the selection was found
1	Syntax error
2	Any other error

Options

-c *count*	For a count of *n*, find and copy the *n*th instance of *selection*.

Examples

```
Copy §
```

Copy the current selection to the Clipboard. This command is like the Copy command in the Edit menu, except that the action takes place in the target window.

```
Copy /BEGIN/:/END/
```

Select everything from the next BEGIN through the following END, and copy this selection to the Clipboard.

See also

Cut and Paste commands

"Selections" in Chapter 4 and Appendix B

Count — count lines and characters

Syntax

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

Description

Counts the lines and characters in its input, and writes the results to standard output. If no files are specified, standard input is read. If more than one file is specified, separate counts are printed for each file, one per line, preceded by the filename, and a total is printed following the list.

Input

Standard input is read if no files are specified on the command line.

Output

Line and character counts are written to standard output.

Diagnostics

Errors are written to diagnostic output.

Status

Count returns the following status values:

0	No errors
1	Error in parameters
2	Unable to open input file

Options

-l	Write only the line counts.
-c	Write only the character counts.

Examples

```
Count MakeFile.c Count.c
```

will display line counts and character counts in the form

```
MakeFile.c  43     981
Count.c     153    3327
Total       196    4303
```

```
Files | Count -l
```

Display the total number of files and directories in the current directory.

```
Count -l §
```

Display the number of lines selected in the target window.

Note The source code for Count is included in the CExamples folder, in the file Count.c, as part of MPW C.

Cut — copy selection to Clipboard and delete it

Syntax

Cut [-c *count*] *selection* [*window*]

Description

Finds *selection* in the specified window, copies its contents to the Clipboard, and then deletes the selection. If no window is specified, the command operates on the target window (the second window from the top). It's an error to specify a window that doesn't exist.

For a definition of *selection*, see "Selections" in Chapter 4; a summary of the selection syntax is contained in Appendix B.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Cut returns the following status values:

0	At least one instance of the selection was found
1	Syntax error
2	Any other error

Options

-c *count*	Finds and cuts *count* instances of *selection*.

Examples

```
Cut §
```

Cut the current selection in the target window. (This is the same as the Cut menu item, except that it operates on the target window rather than the active window.)

```
Cut  /BEGIN/:/END/
```

Select everything from the next BEGIN through the following END, copy the contents of the selection to the Clipboard, and then delete the selection.

See also

Clear, Copy, and Paste commands

"Selections" in Chapter 4 and in Appendix B

CvtObj — convert Lisa Workshop object files to MPW object files

Syntax

CvtObj [-n *namesFile*] [-o *outputFile*] [-p] *LisaObjFile*

Description

Converts a Lisa object file (.OBJ file) to the Macintosh object format (.o file). This command is for Lisa Workshop users who have old object files but no source files that can be ported to the MPW system.

CvtObj supports object files produced by the Lisa Pascal Compiler, the Green Hills C Compiler, and the TLA Assembler that were targeted to the Macintosh run-time environment. Object files produced by other compilers have not been tested, but should work. The program should not be used to convert object files targeted for execution on Lisa.

Object files produced by the Lisa Pascal Compiler must have been compiled with the Macintosh code generation option, $M+. Object files produced by the Green Hills C Compiler must have been compiled with the default code generation option, that is, the **-lisa** option must not have been specified. Assembler code produced by the TLA Assembler should conform to the guidelines outlined in the "Using Assembly Language" chapter of ***Inside Macintosh***.

CvtObj detects and rejects a number of Lisa object record types. If this happens, CvtObj generates a fatal error message ("Can't handle ..."), and terminates without producing an output file. However, CvtObj cannot detect and reject all object files targeted for execution on the Lisa, especially Pascal and TLA Assembler files.

The Lisa Workshop tools support only 8-character case-insensitive (shifted to uppercase) external identifiers. The MPW compilers support variable-length, case-sensitive external identifiers. (The MPW Pascal Compiler still defaults to upshifting Pascal identifiers, primarily for language compatibility, portability of sources, and ease in providing both C and Pascal interfaces to the Macintosh ROM routines.) CvtObj provides the **-n** option for substituting names, so that old object files can be properly linked with new object files. The **-n** option specifies a "names" file, which controls name substitution.

Data initialization. In general, CvtObj automatically matches the Lisa object file semantics with those of the Macintosh. However, data initialization records are more difficult to handle. With the Lisa tools, data areas were often defined with differing lengths, partial contents in different files, and so on. The underlying model was Fortran-named common areas, with multiple initialization sources. On the Macintosh, the default is to use only the first definition of a data module. In order to match the Macintosh default as closely as possible, CvtObj does not emit a defining instance of a data area unless initialization values are seen. For C data areas that need to be initialized to zero, this behavior can result in Linker error messages reporting that the data area names are "unresolved external references." If the references come from a file produced by CvtObj, then the `define` directive can be used in a names (**-n**) file to request CvtObj to emit a defining instance—this should result in a proper size definition for the data area, unless the data area was defined elsewhere as larger.

Note: The DumpObj command can be useful in tracking down and fixing anomalies in external names and data area definitions when using CvtObj.

Input

None.

Output

If no **-o** option is specified, output is written to the file CvtObj.out.o.

Diagnostics

Errors and warnings are written to the diagnostic file. Progress information is also written to the diagnostic file (with the **-p** option).

Status

The following status values are returned:

0	No problem
2	Fatal error
3	User interrupt

Options

-n *namesFile*	Name conversion file. In this text file, lines that begin with a space or tab are interpreted as name substitution lines; the first name is the old name, the second name is the new name. (See "Examples" below.) All occurrences of the old name are replaced with the new name. Lines that begin with the word `define`, followed by an entry name, create a global data module for that name.
-o *outputFile*	Direct output to *outputFile*. The default output filename is CvtObj.out.o.
-p	Write progress information to diagnostic output.

Examples

```
CvtObj -o MyFile.o MyLisaFile.OBJ
```

Convert file MyLisaFile.OBJ, placing the output in MyFile.o.

```
CvtObj -n NewNames -o MyFile2.o  MyLisaFile2.OBJ
```

Convert file MyLisaFile2.OBJ, placing the output in MyFile2.o, and applying the name translations specified in NewNames. The NewNames file might contain the following:

```
ΔCLOSEOUT  CloseOutput
ΔDRAWROUN  DrawRoundFigure
ΔFOO2  Foo2
define FOO
```

where Δ indicates a leading space or tab character.

See also

TLACvt, Link, and DumpObj commands

Appendix H, "Object File Format"

Date — write the date and time

Syntax

Date [-a | -s] [-d | -t]

Description

Writes the current date and time to standard output.

Input

None.

Output

Standard output.

Diagnostics

Errors are written to diagnostic output.

Status

Status code 0 is returned if the options are consistent; otherwise 1 is returned.

Options

-a	Abbreviated date. Three-character abbreviations are used for the month and day of the week. For example, `Thu, Aug 29, 1985`.
-d	Write the date only.
-s	Short date form. Numeric values are used for the date. The day of the week is not given. For example, `8/29/85` (month/day/year).
-t	Write the time only.

Examples

```
Date
```
returns the date in the form
```
Friday, February 14, 1986 10:34:25 PM
```

```
Date -a
```
returns
```
Fri, Feb 14, 1986 10:34:25 PM
```

```
Date -s -d
```
returns
```
2/14/86
```

Delete — delete files and directories

Syntax

```
Delete [ -y | -n ] [ -i ] [ -p ] name...
```

Description

Deletes file or directory *name*. If *name* is a directory, then *name* and its contents (including all subdirectories) are deleted.

For deleting directories, a dialog box will request confirmation for the deletion. The **-y** or **-n** options can be used in command files to avoid this interaction.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output. Progress and summary information is also written to diagnostic output if the **-p** option is specified.

Status

The following status codes are returned:

0	All specified objects were deleted (except for any directories skipped with the **-n** option)
1	Syntax error
2	An error occurred during the delete

Options

-i	Ignore errors (that is, do not print messages, and return a status value of 0).
-n	Answer "no" to any confirmation dialog that may occur, skipping the delete for any directories encountered.
-p	List progress information as the delete takes place.
-y	Answer "yes" to any confirmation dialog that may occur, causing any directory encountered to be deleted.

Examples

```
Delete HD:MPW:≈.c
```

Delete *all* items in the MPW folder that end in ".c". (Recall that the Shell first replaces the parameter "≈.c" with a list of filenames matching the pattern—the Delete command then deletes each of these files.)

Warnings

Beware of potentially disastrous typing mistakes such as the following:

```
Delete ≈ .c
```

Note that the space after "≈"—this space causes "≈" and ".c" to be treated as two separate parameters. In this case, Delete would delete *all files* in the current directory, and also attempt to delete a file named ".c".

Also note that the following command deletes *everything*.

```
Delete ≈:
```

That is, the filename pattern `≈:` expands to the names of *all volumes online* (including the startup volume!).

When deleting files *en masse*, it's a good practice to use the Echo command to verify the action of the filename generation operators; for example,

```
Echo ≈.c
```

See Also

Clear command (for deleting selections)

"Filename Generation" in Chapter 3

DeleteMenu — delete user-defined menus and items

Syntax

DeleteMenu [***menuName*** [***itemName***]]

Description

Deletes the user-defined item ***itemName***, in the menu ***menuName*** . If ***itemName*** is omitted, all user-defined items for ***menuName*** are deleted.

Caution: If ***itemName*** and ***menuName*** are both omitted, all user-defined items are deleted.

Menu items that haven't been added with AddMenu can't be deleted with DeleteMenu.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

DeleteMenu returns the following status values:

0	No errors
1	Syntax error
2	Other errors

Examples

```
DeleteMenu Search
```

Deletes all user-defined items from the Search menu.

See also

AddMenu command

DeRez — Resource Decompiler

Syntax

DeRez [*option*...] *resourceFile* [*resourceDescriptionFile*...]

Description

Creates a text representation (resource description) of the resource fork of *resourceFile*, according to the resource type declarations in the resource description file(s). The resource description is written to standard output.

A **resource description file** is a file of type declarations in the same format as that used by the Resource Compiler, Rez. The type declarations for standard Macintosh resources are contained in the files Types.r and SysTypes.r, contained in the {RIncludes} folder. If no resource description file is specified, the output consists of `data` statements giving the resource data in hexadecimal form, without any additional format information.

If the output of DeRez is used as input to Rez, with the same resource description files, it produces the same resource fork that was originally input to DeRez. DeRez is not guaranteed to be able to run a declaration backwards—if it can't, it produces a `data` statement instead of the appropriate `resource` statement.

DeRez ignores all `include` (but not `#include`), `read`, `data`, and `resource` statements found in the *resourceDescriptionFile*. (It still parses these statements for correct syntax.)

For the format of resource type declarations, see Chapter 6 and Appendix D.

Input

Standard input is never read. DeRez requires a resource file as input. Optional formatting information may be given by specifying one or more resource description files.

For all input files on the command line, the following search rules are applied:

1. DeRez tries to open the file with the name specified "as is."
2. If rule 1 fails, and the filename contains no colons or begins with a colon, DeRez appends the filename to each of the pathnames specified by the {RIncludes} variable and tries to open the file.

Output

A resource description is written to standard output. The resource description consists of `resource` and `data` statements that can be understood by Rez. (See Chapter 6.)

Diagnostics

If no errors or warnings are detected, DeRez runs silently. Errors and warnings are written to diagnostic output.

Status The following status values are returned:

0	No errors
1	Error in parameters
2	Syntax error in file
3	I/O or program error

Options

-d[efine] *macro*[=*data*]

Define the macro variable *macro* to have the value *data*. If *data* is omitted, then *macro* is set to the null string—note that this still means that *macro* is defined. The **-d** option is the same as writing

```
#define macro [ data ]
```

at the beginning of the input. The **-d** option may be repeated any number of times.

-m[axstringsize] *n*

Set the maximum string size to *n*; *n* must be in the range 2–120. This setting controls how wide strings will be in the output.

-only *typeExpr* [(*ID1* [: *ID2*]) | *resourceName*]

Read only resources of resource type *typeExpr*. If an ID, range of IDs, or resource name is given, read only those resources for the given type. This option may be repeated.

Note: *typeExpr* is an expression, so literal quotes (') might be needed. If an ID, range of IDs, or name is given, the entire option parameter must be quoted; for example,

```
DeRez -only "'MENU'(1:128)"...
```

See also the "Examples" section below.

Note: The **-only** option cannot be specified together with the **-skip** option.

-only *type* A simpler version of the above option—no quotes are needed to specify a literal type as long as it starts with a letter. No escape characters or anything fancy is allowed. For example,

```
DeRez -only MENU...
```

-p Display progress and version information.

-rd Suppress warning messages if a resource type is redeclared.

-s[kip] *typeExpr* [(*ID1* [: *ID2*]) | *resourceName*]
: Skip resources of type *typeExpr* in the resource file. For example, it's very useful to be able to skip 'CODE' resources. *typeExpr* is an expression—see the note under **-only**. The **-s** option may be repeated any number of times.

-s[kip] *type*
: A simpler version of the **-s** option—no quotes are needed to specify a literal as long as it starts with a letter.

-u[ndef] *macro*
: Undefine the macro variable *macro*. This is the same as writing

```
#undef macro
```

at the beginning of the input file. It is only meaningful to undefine the preset macro variables. This option may be repeated.

Examples

```
DeRez "{ShellDirectory}MPW Shell" -only MENU  Types.r
```

Display all of the 'MENU' resources used by the MPW Shell. The type definition for 'MENU' resources is found in the file Types.r.

```
DeRez HD:OS:System SysTypes.r  ∂
   -only "'DRVR' (∂"\0x00Scrapbook∂")"
```

Decompile the Scrapbook desk accessory in the copy of the System file that's located in directory HD:OS:. (The type definition for 'DRVR' resources is found in the file SysTypes.r.

See also

Rez and RezDet commands

Chapter 6, "Using the Resource Compiler and Decompiler"

Type declaration files in RIncludes folder:

- Types.r
- SysTypes.r
- MPWTypes.r

Chapter 5, "Editing Resources With ResEdit"

Directory — set or write the default directory

Syntax

Directory [-q | *directory*]

Description

If specified, *directory* becomes the new default directory; otherwise the pathname of the current default directory is written to standard output.

Note: To display a directory's contents, use the Files command.

Input

None.

Output

If no directory is specified, the default directory pathname is written to standard output.

Diagnostics

Errors are written to diagnostic output.

Status

Status code 0 is returned if the command succeeded; otherwise 1 is returned.

Options

-q	Don't quote the pathname that is written to standard output. (Normally, a directory name is quoted if it contains spaces or other special characters.)

Examples

```
Directory
```

Write the pathname of the current directory to standard output.

```
Directory HD:MPW:AExamples:
```

Set the default directory to the folder AExamples in the folder MPW on the volume HD. The final colon is optional.

```
Directory Reports:
```

Set the default directory to the volume Reports. (Note that volume names must end in a colon.)

```
Directory :Include:Pascal:
```

Set the default directory to the folder Pascal in the folder Include in the current default directory.

See also "File and Window Names" in Chapter 1
Files and NewFolder commands

DumpCode — work formatted resources

Syntax

DumpCode [*option...*] *resourceFile*

Description

Disassembles object code that is stored in resources such as 'CODE', 'DRVR', and 'PDEF'. DumpCode reads from the resource fork of the specified file, and writes the formatted assembly code to standard output. The default formatting convention is to disassemble the code, and to display the corresponding bytes in hexadecimal and ASCII.

The default behavior of DumpCode is to dump all the 'CODE' resources from a program file. The **-rt** option can be used to dump resources of other types such as drivers and desk accessories.

Some conventions about executable code resources are built into DumpCode, and affect the formatted output in special ways:

- 'CODE' resources with ID 0 are formatted as a jump table (unloaded format).
- Other 'CODE' resources have information about jump table entries in the first four bytes.
- 'DRVR' resources have a special format at the beginning of the resource.

In addition, you can direct DumpCode to give a symbolic dump of data initialization descriptors and initial values.

Input

None.

Output

DumpCode writes formatted resources to standard output.

Diagnostics

Errors and warnings are written to diagnostic output. Progress information can also be written to diagnostic output (with the **-p** option).

Status

DumpCode returns the following status values:

0	No problem
2	Fatal error
3	User interrupt

Options

Note: Numeric values for options can be specified as decimal constants, or as hex constants preceded by a "$".

-d Suppress the disassembly and dumping of code. (The default is to disassemble the code.)

This option is useful in producing a small output file, and looking at just the resource names, sizes, and resource header information. It is also useful when just some specialized information is desired, such as the jump table or data descriptors.

-h Suppress the writing of header information, such as resource relative locations, hexadecimal and ASCII equivalents, and so on. The default is to produce this type of header information.

This option is useful in producing output that can be edited and submitted to the Assembler for reassembly.

-jt Suppress formatting of jump table. Only summary information for the jump table is given. (The default is to format the jump table unless one of the options **-s**, **-rt**, **-n**, or **-jt** is specified.)

-n Write only the resource names associated with resources. This option is useful for finding segments or desk accessories by name.

-p Write progress information (filenames, resource names, IDs, and sizes) to diagnostic output.

-r *byte1*[,*byteN*] Limit the disassembly of code to the range *byte1...byteN*. The default is to disassemble all bytes in a segment. If *byteN* is omitted, then the rest of the segment is disassembled.

-rt *type*[=*ID*] Dump only the single resource with type *type* and ID number of *ID*. If *ID* is omitted, then all resources of the specified type are dumped.

-s *resourceName*

Dump only the single resource named *resourceName*.

Examples

```
DumpCode Sample > SampleDump
```

Format the 'CODE' resources in the file Sample, writing the output to the file SampleDump. This output has the following format:

```
File: sample, Resource 3, Type: CODE, Name: _DataInit
Offset of first jump table entry: $00000018
Segment is $000000D2 bytes long,  and uses 1 jump table entry
000000: 48E7 FFF0    'H...'       MOVEM.L D0-D7/A0-A3,-(A7)
000004: 4247         'BG'         CLR.W   D7
000006: 4EAD 0032    'N..2'       JSR     $0032(A5)
00000A: 2218         '".'         MOVE.L  (A0)+,D1
```
etc.

See also DumpObj command

"The Jump Table" in the "Segment Loader" chapter of *Inside Macintosh,* for a description of the jump table

DumpObj — write formatted object file

Syntax

DumpObj [*option*…] *objectFile*

Description

Disassembles object code that is stored in the data fork of an object file. By convention, object files end in the suffix **.o**. In addition, the object file must have type 'OBJ '.

Input

DumpObj does not read standard input.

Output

DumpObj writes formatted object file records and disassembled code to standard output.

Diagnostics

Errors and warnings are written to diagnostic output. Progress information is also written to diagnostic output with the **-p** option.

Status

DumpObj returns the following status values:

0	No problem
2	Fatal error
3	User interrupt

Options

-d Suppress disassembly of code and display of data. The default is to disassemble code and to display data in hexadecimal and ASCII.

-i Suppress substitution of names for IDs. The default is to preread the entire file, processing the Dictionary records, and then to show names in place of ID numbers.

This option is useful in examining an object file up to the point where an object file format error has been reported by Link or Lib; that is, it suppresses the preread, which is also likely to fail.

-h Suppress printing of header information on code lines. Header information includes the offset of the code and the code bytes in hex and ASCII. The default is to print header information.

This option is useful in producing code that can be edited and submitted to the Assembler for reassembly.

-l	Print file locations of object records. The default is not to print these locations. This option is useful in debugging compilers and assemblers, particularly when debugging code used to generate Pad records to assure alignment. (See Appendix H, "Object File Format.")
-m *name*	Dump a particular module. If *name* is an entry point, then the module containing *name* is dumped. Other options that control format still have an effect. *Note:* *name* is case sensitive, as are all object file identifiers.
-n	Print names only. When this option is specified, only the **-p** option has an effect. This option is useful in determining which names are defined in an object file, particularly when there appears to be a discrepancy in spelling, capitalization, or length of identifiers.
-p	Write progress information (such as the name of the file being dumped and the version of DumpObj) to diagnostic output.
-r *byte1*[*,byteN*]	Limit the disassembly of code to the range *byte1...byteN*. The default is to disassemble all bytes in a segment. If *byteN* is omitted, then the rest of the segment is disassembled.

Example

```
DumpObj Sample.p.o >SampleDump
```

Formats the file Sample.p.o and writes its contents to the file SampleDump. This output has the following format:

```
Dump of file sample.p.o
First:        Kind 0 Version 1
Dictionary:  FirstId 2
       2: Main
Pad
Module:     Flags $00 ModuleId   1 SegmentId Main
Content:    Flags $00
Contents offset 0000 size 006A
000000: 4E56 FFFE     'NV..'     LINK       A6,#$FFFE
000004: 2F07          '/.'       MOVE.L     D7,-(A7)
000006: 42A7          'B.'       CLR.L      -(A7)
000008: 3F3C 0080     '?<..'     MOVE.W     #$0080,-(A7)
```
etc.

For more information, see Appendix H, "Object File Format."

See also DumpCode command
Appendix H, "Object File Format"

Duplicate — duplicate files and directories

Syntax

```
Duplicate [ -y | -n ] [ -d | -r ] [ -p ] name... targetName
```

Description

Duplicate *name* to *targetName*. (*Name* and *targetName* are file or directory names.) If *targetName* is a file or doesn't exist, then the file or directory *name* is duplicated and named *targetName*. If *targetName* is a directory, then the objects named are duplicated into that directory. (If more than one *name* is present, *targetName* must be a directory.) Created objects are given the same creation and modification dates as their source.

If a directory is duplicated, then its contents (including all subdirectories) are also duplicated. No directory duplicated can be a parent of *targetName*.

Name can also be a volume; if *targetName* is a directory, then *name* is copied into *targetName*.

A dialog box requests a confirmation if the duplicate would overwrite an existing file or folder. The -y or **-n** options can be used in command files to avoid this interaction.

Input

None.

Output

None.

Diagnostics

Progress and summary information is written to diagnostic output if the **-p** option is specified.

Status

The following status codes are returned:

0	All objects were duplicated
1	Missing or inaccessible parameters
2	An error occurred

Options

-y	Answer "yes" to any confirmation dialog that occurs, causing conflicting files or folders to be overwritten.
-n	Answer "no" to any confirmation dialog that occurs, skipping files or folders that already exist.

-d	Duplicate the data fork only. If *targetName* is an existing file, its data fork is overwritten and its resource fork remains untouched.
-r	Duplicate the resource fork only. If *targetName* is an existing file, its resource fork is overwritten and its data fork remains untouched.
-p	List progress information.

Examples

```
Duplicate Aug86 "Monthly Reports"
```

Assuming "Monthly Reports" is an existing directory, duplicate the file Aug86 into that directory.

```
Duplicate File1 Folder1 "Backup Disk:"
```

Duplicate File1 and Folder1 (including its contents) onto Backup Disk.

```
Duplicate -y File1  File2
```

Duplicate File1 to File2, overwriting File2 if it exists.

```
Duplicate Disk1:≈  HD:Files:
```

Duplicate all of the files on Disk1 into the directory HD:Files.

```
Duplicate Disk1:  HD:Files:
```

Duplicate all of Disk1 (as a directory) into HD:Files.

Limitations

Duplicate doesn't recognize folders on non-HFS disks.

See also

Move and Rename commands
"File and Window Names" in Chapter 1
"Filename Generation" in Chapter 3

Echo — echo parameters

Syntax

Echo [-n] [*parameters* ...]

Description

Writes its parameters, separated by spaces and terminated by a return, to standard output. If no parameters are specified, only a return is written.

Echo is especially useful for checking the results of variable substitution, command substitution, and filename generation.

Input

None.

Output

Parameters are written to standard output.

Diagnostics

None.

Status

Status value 0 is always returned.

Options

-n	Don't write a return following Echo's parameters (that is, the insertion point remains at the end of the output line). The **-n** isn't echoed.

Examples

```
Echo "Use Echo to write progress info from command files."
Use Echo to write progress info from command files.
```

The Echo command above writes the second line to standard output.

```
Echo {Status}
```

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

```
Echo ≈.a
```

Echoes the names of all files in the current directory that end with ".a". (This might be useful as a precaution before executing another command with the argument "≈.a".)

```
Echo -n > EmptyFile
```

If EmptyFile exists, this command deletes its contents; if the file doesn't exist, it is created.

See also Parameters command

Eject — eject volumes

Syntax

Eject [-m] *volume*...

Description

Flushes the volume, unmounts it, and then ejects it, if it is a floppy disk. A volume name must end with a colon (:). If *volume* is a number without a colon, it's interpreted as a drive number.

Note: If you unmount the current volume (the volume containing the current directory), the boot volume becomes the current volume. You can keep the volume mounted with the **-m** option. (See the "File Manager" chapter of *Inside Macintosh.*)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status codes are returned:

0	The disk was successfully ejected
1	Syntax error
2	An error occurred

Options

-m	Leave the volume mounted.

Examples

```
Eject Memos:
```

Eject (and unmount) the disk titled Memos.

```
Eject 1
```

Eject and unmount the disk in drive 1 (the internal drive).

See also

Mount, Unmount, and Volumes commands

Entab — convert runs of spaces to tabs

Syntax

Entab [*option* ...] [*file* ...]

Description

Copies the specified text files to standard output, replacing runs of spaces with tabs. The default behavior of Entab is to do the following:

1. Detab the input file using the file's tab setting (a resource saved with the file by the Shell editor), or 4 if there is none. You can override this "detab" value with the **-d** option.
2. Then entab the file, setting tab stops every 4 spaces. You can specify another tab setting with the **-t** option. The entabbed output file looks the same as the original file(s), but contains fewer characters.

Options are also provided for controlling the processing of blanks between quoted strings.

Input

If no filenames are specified, standard input is processed.

Output

All files are written to standard output.

Diagnostics

Parameter errors and progress information (with the **-p** option) are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

-d *tabSetting* Override the input file's default tab setting with *tabSetting*. This option is useful for detabbing non-MPW files.

Note: Entab's default action is to detab the input file, using the file's tab setting, or 4 if there is none. For MPW files, specifying a **-d** option would override the file's own tab setting, leading to incorrect results if a different value were used.

-l *quote...* Specify a list of left quote characters. *Quote...* is a string of one or more nonblank characters. If **-l** is specified, then **-r** must also be specified. Single quotes (') and double quotes (") are assumed as the default quoting characters.

-n Treat all quotes as "normal" characters—entab the file, replacing runs of spaces embedded in quoted strings with tabs.

Caution: This option should not be used when entabbing program source files. If this option is used, the **-q**, **-l**, and **-r** options are ignored.

-p Write version and progress information to diagnostic output.

-q *quote...* Specify a list of characters to be used as both left and right quotes. *Quote...* is a string of one or more nonblank characters. This is the default option; single quotes (') and double quotes (") are assumed as the quoting characters.

-r *quote...* Specify a list of right quote characters. *Quote...* is a string of one or more nonblank characters. If **-r** is specified, then **-l** must also be specified.

Note: Entab does not check that a particular left quote character matches a particular right quote character.

-t *tabSetting* Set the output file's tab setting to *tabSetting*. If the **-t** option is omitted, 4 is assumed for the tab setting. If you specify a tab setting of 0, no tabs are placed in the output. Thus **-t 0** may be used to completely detab input files.

Caution: If you specify the **-q**, **-l**, or **-r** options, then you should quote the entire string parameter to these options (otherwise, the Shell may misinterpret special characters in the parameter string).

Example

```
Entab -t 2 Example.p > CleanExample.p
```

Detab the file Example.p (using the file's default tab setting), re-entab it with a tab setting of 2, and write the resulting output to CleanExample.p.

Caution: Beware of command formats such as

```
Entab Foo > Foo
```

Limitations

Entab does not take into account embedded formatting characters except for tab characters. Thus backspace characters may cause incorrect results.

The maximum width for an input line is 255 characters.

See also Tab command

Equal — compare files and directories

Syntax

Equal [*option*…] *name*… *targetName*

Description

Compares *name* to *targetName*. By default, Equal makes no comment if files are the same; if they differ, it announces the byte at which the difference occurred. When comparing directories, the default condition is to report all differences, including files not found—the -i option ignores files in *targetName* that are not present in *name*.

If *targetName* is a file, every *name* must also be a file. The specified files are compared with *targetName*.

If *targetName* is a directory and *name* is a file, Equal checks in *targetName* for the file *name* and compares the two files. That is, the command

```
Equal File1 Dir1
```

compares File1 with :Dir1:File1.

If more than one name is specified, Equal compares each name with the corresponding file or directory in *targetName* (all subdirectories are also compared). The command

```
Equal File1 Dir1 Dir2
```

compares File1 with :Dir2:File1 and then compares Dir1 with :Dir2:Dir1.

If *targetName* is a directory, *name* is a directory, and only one name is specified, then the Equal command directly compares the two directories. That is, the command

```
Equal Dir1 Dir2
```

compares Dir1 (and all subdirectories) with Dir2.

Input

None.

Output

Differences are written to standard output.

Diagnostics

Errors are written to diagnostic output.

Status

The following status codes are returned:

0	Identical files
1	Syntax error
2	Inaccessible or missing parameter
3	Files not equal

Options

-i	Ignore files missing from directory ***name***, that is, if files in ***targetName*** are not present in ***name***, Equal won't report the missing files as differences.
-d	Compare the data forks only.
-r	Compare the resource forks only.
-p	List progress information as files are compared.
-q	Remain quiet about differences; return status codes only.

Examples

```
Equal File1 File1Backup
```

Report if the files are different and at what point they differ, in a message such as:

```
File1 File1Backup differ in data fork, at byte 5
```

```
Equal -i  HD:Dir1  Disk1:Dir1
```

Compare all files and directories in HD:Dir1 with files and directories with the same names found in Disk1:Dir1, and report any differences. This command does not report files in Disk1:Dir1 that aren't found in HD:Dir1.

```
Equal -i -d Backup:  HD:Source
```

Compare the data forks of all files on the volume Backup: with all those of the same name in the directory HD:Source.

```
Equal -p Old:≈.c  HD:Source
```

Compare all files on Old: ending in .c with their counterparts in HD:Source. Print progress information as the comparison proceeds.

See also

Compare command (Compare is a more elaborate tool that writes two text files' differences to standard output.)

Erase — initalize volumes

Syntax

Erase [-y] [-s] *volume*...

Description

Initializes the specified volumes— the previous contents are destroyed. A volume name must end with a colon (:). If ***volume*** is a number without a colon, it's interpreted as a disk drive number.

A dialog box requests confirmation before proceeding with the command, unless the **-y** option is specified. The **-y** option can be used in command files to avoid this interaction.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status codes are returned:

0	Successful initialization
1	Syntax error
2	No such volume, or boot volume
3	Errors during the initialization procedure

Options

-y	Answer "yes" to the confirmation dialog, causing initialization to begin immediately.
-s	Format the disk for single-sided use (that is, as a 400K, non-HFS disk).

Examples

```
Erase Reports:
```

Initialize the volume titled Reports.

```
Erase 1
```

Initialize the volume in drive 1 (the internal drive). It will be formatted as a 400K disk if drive 1 is a 400K drive, or as an 800K disk if drive 1 is an 800K drive.

Evaluate — evaluate an expression

Syntax

Evaluate [*word...*]

Description

The list of words is taken as an expression. After evaluation, the result is written to standard output. Missing or null parameters are taken as zero. You should quote string operands that contain blanks or any of the characters listed in the table below.

The operators and precedence are mostly those of the C language; they're described below.

Expressions. An expression can include any of the following operators. (In some cases, two or three different symbols can be used for the same operation.) The operators are listed in order of precedence—within each group, operators have the same precedence.

	Operator			**Operation**
1.	(*expr*)			Parentheses are used to group expressions.
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
	>			Greater than
	>=	≥		Greater than or equal
7.	==			Equal
	!=	<>	≠	Not equal
	=~			Equal—regular expression
	!~			Not equal—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. Parentheses can be used 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.

The logical operators !, NOT, ¬, &&, AND, | |, and OR interpret null and zero operands as false and nonzero operands as true. Relational operators return the value 1 when the relation is true, and the value 0 when the relation is false.

The string operators ==, !=, =~, and !~ compare their operands as strings. All others operate on numbers. Numbers may be either decimal or hexadecimal integers representable by a 32-bit signed value. Hexadecimal numbers begin with either $ or **0x**. Every expression is computed as a 32-bit signed value. Overflows are ignored.

The pattern-matching operators =~ and !~ are like == and != except that the right-hand side is a regular expression which is matched against the left-hand operand. Regular expressions must be enclosed within the regular expression delimiters /…/. Regular expressions are summarized in Appendix B.

Note: There is one difference between using regular expressions after =~ and !~ and using them in editing commands—when evaluating an expression that contains the tagging operator, ®, the Shell creates variables of the form {®*n*}, containing the matched substrings for each ® operator. (See the examples below.)

Filename generation, conditional execution, pipe specifications, and input/output specifications are disabled within expressions, to allow the use of many special characters that would otherwise have to be quoted.

Expressions are also used in the If, Else, Break, Continue, and Exit commands.

Input

None.

Output

The result of the expression is written to standard output. Logical operators return the values 0 (false) and 1 (true).

Note: To redirect Evaluate's output (or diagnostic output), you'll need to enclose the Evaluate command in parentheses; otherwise, the > or ≥ symbols will be interpreted as expression operators, and an error will occur. (See the third example below.)

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0 Valid expression
1 Invalid expression

Examples

```
Evaluate (1+2) * (3+4)
```

Do the computation and write the result to standard output.

```
Set lines `Evaluate {lines} + 1`
```

The Set command increments the value of the Shell variable {lines}—the Evaluate command enclosed in command substitution characters (`...`) is replaced by its output.

```
( Evaluate "{aPathname}" =~ /(([¬:]+:)*)®1≈/ ) > Dev:Null
Echo {®1}
```

These commands examine a pathname contained in the variable {aPathname}, and return the directory prefix portion of the name. In this case, Evaluate is used for its side effect of enabling regular expression processing of a filename pattern. The right-hand side of the expression (`/(([¬:]+:)*)®1≈/`) is a regular expression that matches everything in a pathname up to the last colon, and remembers it as the Shell variable {®1}. Evaluate's actual output is not of interest, so it's redirected to the bit bucket, Dev:Null. (See "Pseudo-filenames" in Chapter 3.) Note that the use of I/O redirection means that the Evaluate command must be enclosed in parentheses so that the output redirection symbol, >, is not taken as an expression operator.

This is a complex, but useful, example of implementing a "substring" function. For a similar example, see the Rename command.

See also

"Structured Commands" in Chapter 3

"Pattern Matching (Using Regular Expressions)" in Chapter 4 and Appendix B

Execute — execute a command file in the current scope

Syntax

Execute *commandFile*

Description

Execute the command file as if its contents appeared in place of the Execute command. This means that variable definitions, exports, and aliases in the command file will continue to exist after it has finished executing. (Normally these definitions, exports, and aliases would be local to the command file.) Any parameters following *commandFile* are ignored. Any parameters to the enclosing command file are available within *commandFile*.

Note: If *commandFile* is not a command file (that is, if it's a built-in command, tool, or application), the command is run as if the word Execute did not appear. Parameters are passed to the command as usual.

Input

None.

Output

None.

Diagnostics

None.

Status

Execute returns the status returned by *commandFile*.

Examples

```
Execute "{ShellDirectory}"Startup
```

Execute the Startup (and UserStartup) command files. This command is useful for testing any changes you've made to the Startup-UserStartup script. Variable definitions, exports, and aliases set in Startup and UserStartup will be available after Startup is done executing.

See also

"Defining and Redefining Variables" in Chapter 3

"The Startup and UserStartup Files" in Chapter 3

Exit — exit from command file

Syntax

Exit [*status*] [If *expression*]

Description

If the *expression* is nonzero (that is, true), Exit terminates execution of the command file in which it appears. When used interactively, Exit terminates execution of previously entered commands. *Status* is a number; if present, it is returned as the status value of the command file; otherwise, the status of the previous command is returned. If the "If *expression*" is omitted, the Exit is unconditional. (For a definition of *expression*, refer to the description of the Evaluate command.)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

If *status* is present, it is returned as the status value of the command file. If the expression is invalid, 1 is returned. Otherwise, the status of the last command executed is returned.

Examples

```
Exit {ExitStatus}
```

As the last line of a command file, this Exit command would return as a status value whatever value had previously been assigned to {ExitStatus}.

See also

Evaluate command (for information on expressions)
"Structured Commands" in Chapter 3
{Exit} and {Status} variables, in "Variables," Chapter 3

Export — make variables available to commands

Syntax

Export [*name*...]

Description

Make the specified variables available to command files and tools. The list of variables exported within a command file is local to that command file. An enclosed command file or tool inherits a list of exported variables from the enclosing command file. (See Figure 3-1 in Chapter 3 for clarification.)

Note: You can make a variable available to all command files and tools by setting and exporting it in the Startup or UserStartup files. (Startup acts as the enclosing command file for all Shell operations.)

If no names are specified, a list of exported variables is written to standard output. (Note that the output of Export is in the form of Export commands.)

Input

None.

Output

If no parameters are given, Export writes a list of exported variables to standard output.

Diagnostics

None.

Status

Export always returns a status value of 0.

Examples

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

Define the variable {AIncludes} as the pathname "{MPW}AIncludes:", and make it available to command files and programs.

See also

Set and Execute commands

"The Startup and UserStartup Files" in Chapter 3

"Exporting Variables" in Chapter 3

FileDiv — divide a file into several smaller files

Syntax

FileDiv [-f] [-n *splitpoint*] [-p] *file* [*prefix*]

Description

FileDiv is the inverse of the Catenate command. It is used to break a large file into several smaller pieces. The input file is divided into smaller files, each containing an equal number of lines determined by the *splitpoint* (default=2000). The last file contains whatever is left over.

There is also an option (-f) for splitting a file only when a form feed character (ASCII $0C) occurs as the first character of a line that is beyond the splitpoint. This option lets you split a file at points that are known to be the tops of pages.

Each group of *splitpoint* lines is written to a file with the name *prefixNN*, where *NN* is a number starting at 01. If the *prefix* is omitted, the input file name is used as the prefix.

Input

An input file must be specified in the command line. Standard input is not used.

Output

FileDiv creates files with names of the form *prefixNN*, where *NN* is a number. (If *prefix* is omitted, the input filename is used as a prefix.) Standard output is not used.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

-f	Split the input file only when at least *splitpoint* lines have been written to the current output file *and* there is a form feed character (ASCII $0C) as the first character of a line. The line containing the form feed becomes the first line in the next output file.
-n *splitpoint*	Split the input file into groups of *splitpoint* lines (or, if the **-f** option was specified, *splitpoint* or more lines). If the **-n** option is omitted, 2000 is assumed.

-p	Write version information and progress information to diagnostic output.

Example

```
FileDiv -f -n 2500 Bigfile
```

Split Bigfile into files of at least 2500 lines; split the file at points where there are form feed characters. The output files have the names Bigfile*NN*, where *NN* is 01, 02, and so on.

Limitations

The maximum length of an input line is 255 characters.

Files — list files and directories

Syntax

Files [*option*…] [*name*…]

Description

For each disk or directory named, Files lists its contents; for each file named, Files writes its name and any other information requested. Information is written to standard output. By default the output is sorted alphabetically. If no name is given, the current directory is listed.

Input

None.

Output

File information is written to standard output.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	All names were processed successfully
1	Syntax error
2	An error occurred

Options

-c *creator* List only those files with the given file creator.

-l List in long format, giving name, type, creator, size, attributes, modification date, and creation date.

The attributes listed under the **-l** option consist of the following characters:

L	Locked
V	Invisible
B	Bundle
S	System
P	Protected
O	Open
I	Inited
D	(on) Desktop

Uppercase letters indicate a value of 1, lowercase a value of 0. See the "File Manager" chapter of ***Inside Macintosh*** for information.

-q	Don't quote names in the output. (Normally, the Files command quotes names that contain spaces or special characters.)
-r	Recursively list subfolders encountered; that is, list every file in every directory.
-t *type*	List only those files with the given file type.

Examples

```
Files {MPW} -t TEXT
```

List all files of type TEXT in the {MPW} directory.

```
Files  -l "MPW Shell"
```

Write "long" output such as

```
                   Name                 Type Crtr  Size   Flags      Last-Mod-Date      Creation-Date
                   -----------------    ---- ----  ----  ---------  -----------------  ---------------
--
                   MPW Shell            APPL MPS   166K lvBspOId    8/8/86   4:51 PM    8/8/86   4:51
PM
```

Find — find and select a text pattern

Syntax

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

Description

Create a selection in *window*. If no window is specified, the target window (the second window from the top) is assumed. It's an error to specify a window that doesn't exist.

Selection is a selection as defined in Chapter 4 and in Appendix B.

Note: Searches do not necessarily start at the beginning of a window—a forward search begins at the end of the current selection and continues to the end of the document. A backward search begins at the start of the current selection and continues to the beginning of the document.

All searches are case insensitive by default. You can specify case-sensitive searches by first setting the Shell variable {CaseSensitive} to a nonzero value. (You can automatically set {CaseSensitive} by selecting the Case Sensitive item from the Find menu.)

Input

None.

Output

None.

Diagnostics

None.

Status

The following status codes are returned:

0	At least one instance of the selection was found
1	Syntax error
2	Any other error

Options

-c *count*	For a count of *n*, find the *n*th occurrence of the selection.

Examples

```
Find •
```

Position the insertion point at the beginning of the target window.

```
Find -c 5 /procedure/ Sample.p
```

Select the fifth occurrence of "procedure" in the window Sample.p.

```
Find 332
```
Select line 332 in the target window.

See also "Selections" and "Pattern Matching" in Chapter 4
"Find Menu" in Chapter 2

Font — set font characteristics

Syntax

Font *fontname* *fontsize* [*window*]

Description

Change the font family and point size of all text in *window* to *fontname* and *fontsize*. Both *fontname* and *fontsize* are required. It's an error to specify a window that doesn't exist. If no window is specified, the command operates on the target window (the second window from the top).

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Font returns the following status values:

0	Successful completion
1	Error in parameters
2	An illegal fontname or fontsize was specified

Examples

```
Font Monaco 12
```

Change the font of the target window to Monaco 12 point.

For... — repeat commands once per parameter

Syntax

For *name* In *word* ...
  command...
End

Description

Executes the list of commands once for each word from the "In *word* ..." list. The current word is assigned to variable *name*, and you can therefore reference it in the list of commands by using the notation {*name*}. Return characters must appear at the end of each line as shown above, or they can be replaced with semicolons (;).

The Break command can be used to terminate the loop. The Continue command can be used to terminate the current iteration of the loop.

The pipe specification (|), conditional command terminators (&& and | |), and input/output specifications (<, >, >>, ≥, and ≥≥) may appear following the End, and apply to all of the commands in the list.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

For returns the following status values:

0	The list of words or list of commands was empty
1	There was an error in the parameters to For

Otherwise, the status of the last command executed is returned.

Examples

```
For i In 1 2 3
   Echo i = {i}
End
```

Returns the following:

```
i = 1
i = 2
i = 3
```

```
For File In ≈.c
   C "{File}" ; Echo "{File}" compiled.
End
```

This example compiles every file in the current directory whose name ends with the suffix ".c". The Shell first expands the filename pattern ≈.c, creating a list of the filenames after the "In" word. The enclosed commands are then executed once for each name in the list. Each time that the loop is executed, the variable {File} represents the current word in the list. {File} is quoted because a filename could contain spaces or other special characters.

```
For file in Startup UserStartup Suspend Resume Quit
   Entab "{file}" > temp
   Rename -y temp "{file}"
   Print -h "{file}"
   Echo "{file}"
End
```

This example entabs (replaces multiple spaces with tabs) the five files listed, prints them with headings, and echoes the name of each file, after printing is complete. You might want to use this set of commands before making copies of the files to give to a friend. Entabbing the files saves considerable disk space, and printing them gives you some quick documentation to go with the files.

See also

Loop, Break, and Continue commands

"Structured Commands" in Chapter 3

Help — display summary information

Syntax

Help [-f *helpFile*] [*command*...]

Description

Help writes information about the specified commands to standard output. If no command is specified, information about Help is written to standard output. *Command* can include any of the following:

commandName	information about *commandName*
`commands`	a list of all MPW commands
`expressions`	a summary of expressions
`patterns`	a summary of pattern specifications (regular expressions)
`selections`	a summary of selection operators
`characters`	a summary of MPW Shell special characters

By default, the Help command looks for information in the file MPW.Help. It looks for this file first in {Shell Directory}; if it isn't found, it looks in {System Folder}.

The following syntax notation is used to describe Macintosh Workshop commands:

[*optional*]	Square brackets mean that the enclosed elements are optional.
repeated...	Ellipses indicate that the preceding item can be repeated one or more times.
a | *b*	A vertical bar indicates an either/or choice.
(*grouping*)	Parentheses indicate grouping (useful with " | " and "...").
< *input*	If *input* is not specified, the command reads from standard input.
> *output*	The command writes to standard output.
≥ *progress*	Progress information is written to diagnostic output (with the **-p** option).

Input

None.

Output

Command information is written to standard output.

Diagnostics

None.

Status

The following status codes are returned:

0 Information could be found for the given command
1 Syntax error
2 A command could not be found, or error in parameters
3 The help file could not be opened

Options

-f helpFile	Specify help file to be searched. (A help file is an ordinary MPW text file.) The default file is MPW.Help.

Example

```
Help Rez
```

Writes information such as

```
Rez [option…] [file…]  < file ≥ progress
     -c[reator] creator       # set output file creator
     -d[efine] name[=value]   # equivalent to  #define macro [value]
     -o file                  # write output to file (default Rez.Out)
     -p                       # write progress information to diagnostic
     -rd                      # suppress warnings for redeclared types
     -ro                      # set the mapReadOnly flag in output
     -t[ype] type             # set output file type
     -u[ndef] name            # equivalent to  #undef name
```

If... — conditional command execution

Syntax

If *expression*
  command...
[Else If *expression*
  command...] ...
[Else
  command...]
End

Description

Executes the list of commands following the first *expression* whose value is nonzero. (Null strings are considered zero.) At most one list of commands is executed. You may specify any number of "Else If" clauses. The final Else clause is optional. The Return characters must appear at the end of each line as shown above, or they can be replaced with semicolons (;).

The pipe specification (|), conditional command terminators (&& and | |), and input/output specifications (<, >, >>, ≥, and ≥≥) may appear following the End word, and apply to all of the commands in the list.

For a definition of *expression*, see the description of the Evaluate command.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

If none of the lists of commands is executed, the If command returns a status value of 0. Otherwise, it returns the value returned by the last command executed.

Examples

```
If {Status} == 0
   Beep  1a,25,200
Else
   Beep -3a,25,200
End
```

Produce an audible indication of the success or failure of the preceding command.

```
For window in `Windows`
   If "{window}" != "{Worksheet}" AND "{window}" != "{Active}
      Close "{window}"
   End
End
```

Close all of the open windows except the active window and the Worksheet window. (Refer also to the Windows command.)

The following commands, as a command file, would implement a trivial case of a general "compile" command:

```
If "{1}" =~ /≈.c/
   C {COptions} "{1}"
Else If "{1}" =~ /≈.p/
   Pascal {POptions} "{1}"
End
```

If the above commands were saved as a command file (say, as "Compile"), both C and Pascal programs could be compiled with the command

`Compile` *filename*

See also

Evaluate command (for a description of expressions)

"Structured Commands" in Chapter 3

Lib — combine object files into a library file

Syntax

Lib [*option*...] *objectFile* ...

Description

Combines the specified object files into a single file. By convention, input files end in the suffix **.o**, which must be present. In addition, input files must have type 'OBJ ' and creator 'MPS '.

Lib is used for the following:

- Combining object code from different languages into a single file.
- Combining several libraries into a single library, for use in building a particular application or desk accessory. This can greatly improve the performance of the Linker.
- Deleting unneeded modules (with the **-dm** option), changing segmentation (the **-sg** and **-sn** options), renaming external modules (the **-mn** option), or changing the scope of a symbol from external to local (the **-dn** option). (These options are useful when you construct a specialized library for linking a particular program.)

Object files that have been processed with Lib result in significantly faster Links when compared with the "raw" object files produced by the Assembler or Compilers.

The output of Lib is logically equivalent to the concatenation of the the input files, except for the optional renaming, resegmentation, and deletion operations, and the possibility of overriding an external name. The resolution of external names in Lib is identical to Link—in fact, the two programs share the same code for reading object files. Although multiple symbols are reduced to a single symbol, no combining of modules into larger modules is performed, and no cross-module references are resolved. This behavior guarantees that the Linker's output will be the same size whether or not the output of Lib was used.

See "Library Construction" in Chapter 7 for a detailed discussion of the behavior and use of Lib.

Input

Lib does not read standard input.

Output

Lib does not write to standard output. The combined library output is placed in the data fork of the output library file. The default output file is **Lib.Out.o**—you can specify another name with the **-o** option. The output file is given type 'OBJ ' and creator 'MPS '.

Diagnostics

Errors and warnings are written to diagnostic output. Progress information is also written to the diagnostic file if you specify the **-p** option.

Status

Lib returns the following status values:

0	No problem
2	Fatal error
3	User interrupt

Options

-b Do a big execution of Lib, that is, **-bf** and **-bs 4** options.

-bf Allow a big number of files; that is, keep only one input file open at a time. If Lib fails with a "too many files open" message, use this option.

-bs ***nn*** Set the buffer size for input to ***nn*** blocks (512 bytes each). If Lib fails with a "heap error" or "out of memory" message, try this option. Values for ***nn*** must be between 2 and 64. (The default is 16.)

Note: Numeric values can be specified as decimal constants, or as hex constants preceded by a "$".

-d Suppress warnings for duplicate symbol definitions (data and code).

-df *deleteFile* Delete the list of external modules found in *deleteFile*. *DeleteFile* is a text file generated by the Linker option **-uf**. See the Link command and "Library Construction" in Chapter 7 for information.

-dm ***name*** [,***name*** ...]

"Delete Module"—delete the specified external modules from the output file. ***name*** may be either an external module or entry-point name. For each entry point ***name***, the entire module containing the entry point is deleted, together with all other entry names in the module. The contents of the module and all entry points are removed from the output file.

Note: References to names deleted in this way will persist as references "by name." That is, if the references are from active code, they'll need to be resolved by external modules or entry points in another file.

The primary use of this operation is to make the library file smaller, so subsequent links are faster. You can use the Linker option **-uf**, which lists unreferenced ("dead") modules or entry points, to generate a list of names that can be deleted in this way.

-dn *name* [,*name* ...]

"Delete Name"—delete the list of external names from the output file, by reducing their scope to local. **-dn** is a "gentle" deletion, in that it affects only the list of *external* module or entry-point names. The contents of the module, other entry points, references, and so on will still be present in the output file. References to names "deleted" in this way will continue to refer to the same code, but with local scope. This is a useful operation when a global name conflict occurs between two pieces of code, one of which is library code from which you don't need to call the name directly.

-mn *oldName*=*newName*

Change the module or entry point named *oldName* to the name *newName*. See the **-ma** option of the Link command for a description of a similiar option.

-o *name*.o — Place the output in file *name*.o. (The default name is Lib.Out.o).

-p — Write progress and summary information to diagnostic output.

-sg *newSeg*=*oldSeg1*[,*oldSeg2*]...

Change segment names. All code in the old segments named *oldSeg1*,*oldSeg2*,... is placed in the segment named *newSeg*.

-sn *oldSeg*=*newSeg*

Change a segment name. All code in the segment named *oldSeg* is placed in the segment named *newSeg*.

Note: The **-sn** and **-sg** options behave exactly as in Link, except that **-sn** is limited to identifiers on both sides of the equal sign. The arbitrary string for a desk accessory name can be introduced only with Link, not with Lib. The major difference between **-sn** and **-sg** is that the order of the option parameters *oldSeg* and *newSeg* is reversed. (This is done for consistency with Link.)

-w — Suppress warning messages.

Examples

```
Lib {CLibraries}≈ -o {CLibraries}CLibrary.o
```

Combine all of the library object files from the {CLibraries} directory into a single library named CLibrary.o. For applications that require most or all of the C library files, using the new CLibrary file will reduce link time.

See also

`Link, DumpObj, and DumpCode commands`

"Optimizing Your Links" and "Library Construction" in Chapter 7

Appendix H, "Object File Format"

Link — link an application, tool, or resource

Syntax

Link [*option...*] *objectFile...*

Description

Links the specified object files into an application, tool, desk accessory, or driver. The input object files must have type 'OBJ ' and creator 'MPS ', and must end in the suffix ".o". Linked segments from the input object files are placed in code resources in the resource fork of the output file. The default output file name is **Link.Out**—you can specify other names with the **-o** option.

For detailed information about the Linker, and instructions for linking applications, MPW tools, and desk accessories, see Chapter 7.

The Linker's default action is to link an application, placing the output segments into 'CODE' resources. When you link an application, all old 'CODE' resources are deleted before the new 'CODE' resources are written. By default, resources created by the Linker are given resource names that are the same as the corresponding segment names. You can change a resource (segment) name with the **-sn** or **-sg** options; you can create unnamed resources with the **-rn** option.

The Linker executes in three phases:

1. Input phase: The Linker reads all input files, finds all symbolic references and their corresponding definitions, and constructs a reference graph. Duplicate references are found and warnings are issued.
2. Analysis phase: The Linker allocates and relocates code and data, detects missing references, and builds the jump table. If the **-l** or **-x** options are given, the Linker produces a linker map or cross-reference listing. The Linker also eliminates unused code.
3. Output phase: The Linker 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. (The cursor spins backward during this phase.)

Input

Link does not read standard input.

Output

By default, linked segments are placed in 'CODE' resources in the resource fork of the output file. The default output file name is Link.Out—you can specify other names with the **-o** option. If the output file already exists, the Linker adds or replaces code segments in the resource fork. If the output file doesn't exist, it is created with file type APPL and creator '????'. The **-t** and **-c** options can be used to set the output file type and output file creator to other values.

Note: If a Linker error or user interrupt causes the output file to be invalid, then the Linker sets the modification date on the file to "zero" (Jan. 1, 1904, 12:00 a.m.). This guarantees that Make will recognize that the file needs to be relinked.

If you specify the **-l** option, the Linker writes a **location map** to standard output. The map is produced in location ordering, that is, sorted by *segNum, segOffset* . The format is divided into several fields:

name *segName* *segNum, segOffset* [@*JTOffset*] [#] [E] [*fileNum, defOffset*]

See Chapter 7 for more information.

Diagnostics

Errors and warnings are written to diagnostic output. Progress information is also written to diagnostic output if the **-p** option is specified.

Status

The following status values are returned:

0	No problem
2	Fatal error
3	User interrupt

Options

Note: Numeric values for options can be specified as decimal constants, or as hex constants preceded by the dollar sign character ($).

-b	Do a big link; that is, do both **-bf** and **-bs 4** options.
-bf	Allow a big number of files; that is, keep only one input file open at a time. If a link fails with a "too many files open" message, use this option.
-bs *blocks*	Set the buffer size for the Linker to *blocks* blocks (512 bytes each). If a link fails with a "heap error" or "out of memory" message, try this option. Values for *blocks* must be between 2 and 64. (The default is 16.)
-c *creator*	Set the output file creator to *creator*. The default creator is '????'.
-d	Suppress warnings for duplicate symbol definitions (for data and code).

-da Convert segment names to desk accessory names on output. Desk accessory names begin with a leading null character ($00). This option is used when linking assembly-language code into a final desk accessory (resource type 'DRVR').

-l Write a location-ordered map to standard output. Usually, this option will be used with output redirection in effect. For example,

```
Link ObjFile -l > MyMapFile
```

-la List anonymous symbols in the location map (with the **-l** option). The default is to omit anonymous symbols from the map.

-lf In the location map data (**-l** option), include the location where symbols are defined in the input file, that is, the file number and byte offset of the Module or Entry-Point record. (See Appendix H, "Object File Format," for more details.) The default is to omit the symbol definition location.

-m *mainEntry* Set (or override) the main entry point specified in the object files. *MainEntry* is a module or entry-point name.

Note: For an application or MPW tool, the main entry point is assigned the first jump-table entry, as required by the Segment Loader. If a main entry point is specified for a desk accessory, driver, or other type of link, for purposes of using the Linker's active-code analysis feature, then the main entry point should be the first byte of code in the first Linker input file. (A desk accessory has no jump table.)

-ma *name=alias* "Module alias"—give the module or entry-point *name* the new name *alias*. This option lets you resolve undefined external symbols at link time, when the problem is caused by differences in spelling or capitalization. Note that you can't use an alias specification to override an existing module or entry point.

Note: You can alias aliases, as long as the chain of aliases is not circular.

-o *outputFile* Place the Linker output in *outputFile*. If no **-o** option is specified, the default output filename is Link.Out.

-p Write progress and summary information to diagnostic output.

-ra [*seg*]=*nn*

Set the resource attributes of a segment or segments. If *seg* is specified, the single segment named *seg* is given the attribute value *nn*. If *seg* is omitted, then all segments except 0 and 1 are given the attribute value *nn*. (If you intend to set the attributes of all segments, then you must specify this option before any other options that name segments, such as **-sn** and **-sg**.) The segment containing the main entry point (the 'CODE' resource with ID=1) must be set individually to override the default resource attributes (described in Chapter 7).

-rn

Suppress the setting of resource names. (The default is to name each resource with the name of the segment.) Desk accessories must always be named.

-rt *type*=*ID*

Set the output resource type to *type* and the ID to *ID*. This option indicates the link of a desk accessory or driver—that is, only one resource is modified. (The default is type 'CODE' and resource IDs numbered from 0.)

Assembly-language note: When you link a desk accessory or driver, the Linker uses PC-relative offsets, and attempts to edit JSR, JMP, LEA, or PEA instructions from A5-relative to PC-relative addressing mode. Other instructions will generate an error message.

-sg *newSeg*=*oldSeg1*[,*oldSeg2*]...

Change segment names. All code in the segments named *oldSeg1*, *oldSeg2*,... is placed in the segment named *newSeg*.

-sn *oldSeg*=*newString*

Change a segment name. All code in the segment named *oldSeg* is placed in the segment named *newString*.

There are two major differences between **-sn** and **-sg**:

- **-sn** allows an arbitrary string for the new name, whereas **-sg** is intended only for identifiers separated by commas.
- The order of the *oldSeg* and *newSeg* parameters is reversed.

For example,

```
Link... ∂
    -sg Main=SAConsol,StdIO,%A5Init   ∂
    -sn Main="MyDA"  ∂
...
```

The first option combines the three specified segments into one segment named Main; the second option renames Main to "MyDA".

-ss *size* — Change the maximum segment size to *size*. The default value is 32760 (32K less a few overhead bytes). The value *size* can be any value greater than 32760.

64K ROM note: Caution! Applications with segments greater than 32K in size may not load correctly on Macintoshes with 64K ROMs.

-t *type* — Set the output file type to *type*. The default type is APPL.

-uf *deleteFile* — List unreferenced modules in the text file *deleteFile*. (This option is useful in identifying dead source code.) This file can be used as input to Lib in building a specialized library that optimizes subsequent links. See the Lib command's **-df** option and "Library Construction" in Chapter 7 for more details.

-w — Suppress warning messages.

Note: Warnings generally indicate potential errors at run time.

-x *crossRefFile* — Generate a cross-reference listing of active modules and entry points. The listing is ordered by module within each segment. For each module, the following information is listed: each active entry point in the module, other modules and entry points that are referenced by the module, and other modules that reference this module. For each entry point in a module, the modules that reference the entry point are listed.

Examples

```
Link Sample.p.o  ∂
     "{PLibraries}"PInterface.o ∂
     "{PLibraries}"Paslib.o ∂
     "{Libraries}"Runtime.o ∂
     -o Sample ∂
     -l -la >Sample.map ∂
```

Link the main program file Sample.p.o with the libraries PInterface.o, PasLib.o, and Runtime.o, placing the output in Sample, and placing the Linker map in the file Sample.map. Sample will be an application, which can be launched from the Finder or executed from MPW.

```
Link  -rt MROM=8  -c 'MPS '  -t ZROM  -ss 140000 ∂
     -l > ROMLocListing  -o MyROMImage  {LinkList}
```

Link the files defined in the Shell variable {LinkList} into a ROM image file, placing the output in the file MyROMImage. The segment size is set to 140,000 bytes, and the ROM is created as a resource 'MROM' with ID=8. The file is typed as being created by MPW (creator 'MPS '), with file type ZROM. The Linker location-ordered listing is placed in the file ROMLocListing.

For additional examples, see "Linking" in Chapter 7 and the makefiles in the Examples folders for the languages you are using.

See also

"Linking" and "More About Linking" in Chapter 7

Lib command, in this chapter

"The Segment Loader," ***Inside Macintosh***, Volume II

"The Resource Manager," ***Inside Macintosh*** , Volume I

Inside Macintosh, Volume IV for information on the 128K ROM, System Folder, and Finder

Appendix H, "Object File Format"

Loop...End — repeat command list until Break

Syntax

```
Loop
  command...
End
```

Description

Executes the enclosed commands repeatedly. The Break command is used to terminate the loop. The Continue command can be used to terminate the current iteration of the loop. Return characters must appear as shown above, or be replaced with semicolons (;).

The pipe specification (|), conditional command terminators (&& and ||), and input/output specifications (<, >, >>, ≥, and ≥≥) may appear following the End word, and apply to all of the commands in the list.

Input

None.

Output

None.

Diagnostics

None.

Status

Loop returns the status of the last command executed.

Example

The command file below runs a command several times, once for each parameter.

```
###  Repeat - Repeat a command for several parameters  ###
#
#  Syntax:
#             Repeat  command  parameter...
#
# Execute command once for each parameter in the parameter
# list. Options can be specified by including them with
# the command name in quotes.
#
Set cmd "{1}"
Loop
   Shift
   Break If "{1}" == ""
   {cmd} "{1}"
End
```

Notice that Shift is used to step through the parameters, and that Break ends the loop when all the parameters have been used.

See also

Break, For, and Continue commands

"Structured Commands" in Chapter 3

Make — build up-to-date version of a program

Syntax

Make [*option*...] [*targetFile*...]

Description

Generates a set of Shell commands that you can execute to build up-to-date versions of the specified target files. Make allows you to rebuild only those components of a program that require rebuilding. Make determines this by reading a **makefile**—this is a text file that describes dependencies among the components of a program, and associates sets of Shell commands with those dependency relations. You can specify makefiles with the **-f** option. After evaluating the makefile, Make writes the appropriate set(s) of commands to standard output.

See "Using Make" in Chapter 7 for a description of the format of a makefile.

Make executes in two phases:

1. In the first phase, Make reads the makefiles (the "beachball" cursor spins backwards during this phase).
2. In the second phase, Make generates the build commands for the target (the cursor spins forwards)—if a target file doesn't exist or if it depends on files that are out-of-date or newer than the target, Make writes out the appropriate command lines for updating the target file.

You can execute the build commands after Make is done executing.

Input

Standard input is not read. If you don't specify a makefile with the **-f** option, Make tries to open a file called MakeFile. If no target file is specified, Make uses the first target encountered in the makefile.

Output

If a file needs to be updated, Make writes a list of Shell commands to standard output.

Diagnostics

Errors and warnings are written to diagnostic output. If you specify the **-p** option, progress and summary information is also written to diagnostic output.

Status

The following status values are returned:

0	Successful completion
1	Parameter or option error
2	Execution error

Options

-d *name*=*value*	Define a variable *name* with the given *value*. Variables defined from the command line take precedence over definitions of the same variable in the makefile. Thus definitions in the makefiles act as defaults which may be overridden from the command line.
-e	Rebuild everything, regardless of whether targets are out of date. This option causes Make to unconditionally output all of the commands to rebuild the specified targets.
-f *makefile*	Read dependency information from *makefile*. You can specify more than one **-f** option—all dependency information is treated as if it were in a single file. (If no **-f** option is specified, the default file is a file named MakeFile in the current directory.)
-p	Write progress information to diagnostic output. (Normally, Make runs silently, unless errors are detected.)
-r	Find roots (that is, the top level) of the dependency graph. (See the **-s** option.)
-s	Show structure of target dependencies. This option writes a dependency graph for the specified targets to standard output, using indentation to indicate levels in the dependency tree. Circular dependencies are noted, if present. *Note:* This option overrides the normal Make output. It's useful in debugging or verifying complicated makefiles.
-t	"Touch" dates of targets and their prerequisites, that is, bring files up to date by adjusting their modification dates, without outputting build commands. This option is used to bring a set of files up to date when they appear not to be, such as when you've only made changes to comments. **-t** does the minimal adjustment needed to satisfy the dependency relationships—files are touched only if required, and are given the date of their newest dependency, to minimize the repercussions of the date adjustments (this feature is especially useful if the touched file is also a prerequisite for other programs). *Note:* This option overrides normal Make output.
-u	Write a list of unreachable targets to diagnostic output (for debugging).

-v Write verbose output to the diagnostic output file. This option is useful if you want to figure out what Make is doing. The diagnostic output will indicate if targets do not exist, whether or not they need to be rebuilt, and why they need to be rebuilt. It also indicates targets in the makefile that were not reached in the build.

-w Suppress warning messages. Warning messages are issued for things such as files with dates in the future and circular dependency relationships.

Example

```
Make -p -f MakeFile.a Sample
```

Make the target file Sample, printing progress information. Sample's dependency relations are described in the makefile MakeFile.a:

```
Sample          ƒ     Sample.r
     Rez Sample.r -o Sample
Sample          ƒƒ    Sample.r Sample.a.o
     Link Sample.a.o -o Sample
Sample.a.o      ƒ     Sample.a
     Asm Sample.a
```

The ƒ (Option-F) character means "is a function of"—that is, the file on the lefthand side depends on the files on the righthand side. If the files on the righthand side are newer, the subsequent Shell commands are written to standard output. (See Chapter 7 for details.)

See also

"Using Make" in Chapter 7, for the format of a makefile, examples, and other information about using Make.

Makefiles for building sample programs are contained in the Examples folders:

- :AExamples:Makefile.a
- :PExamples:Makefile.p
- :CExamples:Makefile.c

MDSCvt — convert MDS Assembler source

Syntax

MDSCvt [*option* ...] [*file* ...]

Description

Converts the specified Macintosh 68000 Development System (MDS) Assembler source files to the syntax required by the MPW Assembler. The following elements are converted:

- tokens within statements
- special tokens within macros
- directives

For a description of these conversions, refer to "MDS Conversion" in Appendix E of the companion manual *MPW Assembler Reference*.

Input

Standard input is converted if no filenames are specified. If the **-main**, **-g**, or **-l** options are used, only one filename should be specified.

Output

If input is from the standard input file, the converted output is written to standard output. If the input file name is Name, the converted output is written to Name.a. The **-n**, **-prefix**, and **-suffix** options let you modify the naming conventions for the output file.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution aborted

Options

-d	Detab the input. All tabs are removed and replaced with spaces. The default setting is 8 spaces; this value can be changed with the **-t** option.

-e	Detab the input (like the **-d** option), and entab the output as a function of the tab setting (either 8, or the value specified with the -t option).
-f *directivesFile*	Set the case (upper/lower) of directives according to the entries in *directivesFile*. The file MDSCvt.Directives is supplied for this purpose; you can edit it to change the capitalization. If you don't use this option, then all directives are converted to uppercase.

-g *globals*	Convert a main program source and reserve globals space below A5. *Globals* may be specified in decimal or hexadecimal (by preceding the value with a $). The value specified must be negative. For example,
	`-g -512`
	`-g $200`

-i	Convert include files. No PROC or MAIN or END will be generated by MDSCvt.
-m	Do *not* insert MDS-compatible mode-setting directives (BLANKS ON and STRING ASIS) into the converted source.
-main	Convert a main program source. The conversion is done to make the file look like the main code and data modules. Only one file should be converted when using this option.
-n	Do not add the ".a" extension to the input filename to produce the output filename. If you use this option, you must also specify either **-prefix** or **-suffix**.)
-p	Write MDSCvt's version information and conversion status to diagnostic output.
-**pre[fix]** *string*	If the input filename is "Name", the output filename is produced by prefixing the *string* to Name, that is, "*string*Name.a". (You can suppress the ".a" suffix by using the **-n** option, or change it by using the **-suffix** option.)
-**suf[fix]** *string*	If the input filename is "Name", the output filename is produced by appending the *string* to the filename, that is, "Name*string*". The default suffix is .a.
-t *tabSetting*	Set the tab value for input and output files to *tabSetting* value (2 to 255). The default setting is assumed to be 8.

-u *c*	When MDSCvt detects a name in the opcode field that is the same as an MPW directive, it appends the character *c* to make the name unique. (The default character is #.)
-l *identifier*	Convert a main program source and define the main program's entry point by the specified identifier. This option corresponds to the MDS Linker's ! command.

Example

```
MDSCvt  -t 8  MDSFile1.Asm  MDSFile2.Asm
```

Convert MDS Assembler source files MDSFile1.Asm and MDSFile2.Asm to MPW Assembler source files MDSFile1.Asm.a and MDSFile2.Asm.a. The **-t** option sets the tab setting for both files to 8, and entabs the output files based on that value. It is assumed that neither file is a main program because the **-main** option has not been specified. If either file is a main program, then the **-main** option should be specified and only that file should be specified as input to MDSCvt.

Limitations

See Appendix E in *MPW Assembler Reference* for details of conversions that can and cannot be done with MDSCvt.

See also

Appendix E, "MDS Conversion," in *MPW Assembler Reference*

Mount — mount volumes

Syntax

Mount *drive*...

Description

Mounts the disks in the specified drives, making them accessible to the file system. *Drive* is the drive number.

Mounting is normally automatic when a disk is inserted. The Mount command is needed for mounting multiple hard disks, which cannot be "inserted," or if a volume has been unmounted via the Unmount command.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	The disk was mounted
1	Syntax error
2	An error occurred

Example

```
Mount 1
```

Mount the disk in drive 1 (the internal drive).

See also

Unmount and Volumes commands

Move — move files and directories

Syntax

Move [-y | -n] [-p] *name*... *targetName*

Description

Moves *name* to *targetName*. (*Name* and *targetName* are file or directory names.) If *targetName* is a directory, then one or more objects (files and/or directories) are moved into that directory. If *targetName* is a file or doesn't exist, then file or directory *name* replaces *targetName*. In either case, the old objects are deleted. Moved objects retain their current creation and modification dates.

If a directory is moved, then its contents, including all subdirectories, are also moved. No directory moved can be a parent of *targetName*.

Name can also be a volume; if *targetName* is a directory, then *name* is copied into *targetName*.

A dialog box requests a confirmation if the move would overwrite an existing file or folder. The **-y** or **-n** options can be used to avoid this interaction.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output. Progress and summary information is also written to diagnostic output if the **-p** option is specified.

Status

The following status values are returned to the Shell:

0	All objects were moved
1	Syntax error
2	An error occurred during the move

Option

-y	Answer "yes" to any confirmation dialog that may occur, causing conflicting files or folders to be overwritten.
-n	Answer "no" to any confirmation dialog that may occur, skipping the move for files or folders that already exist.
-p	List progress information as the move takes place.

Examples

```
Move Startup Suspend Resume Quit "{SystemFolder}"
```

Move the four files from the current directory to the System Folder.

```
Move File ::
```

Move File from the current directory to the enclosing (parent) directory.

```
Move -y File1 File2
```

Move File1 to File2, overwriting File2 if it exists. (This is the same as renaming the file.)

See also

Duplicate and Rename commands
"File and Window Names" in Chapter 1
"Filename Generation" in Chapter 3

New — open a new window

Syntax

New [*name*]

Description

Opens a new window as the active (topmost) window. If *name* is not specified, the Shell generates a unique name for the new window, of the form "Untitled-*n*", where *n* is a decimal number. If *name* already exists, an error results.

Note: New is slightly different from the Open command with the -n option, which simply brings the specified window to the top if it already exists, without returning an error.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

New returns the following status values:

0	No errors
1	Error in parameters
2	Unable to complete operation
3	System error

Examples

```
New Test.a
```

Open a new window named Test.a.

```
New
```

Open a new window with a Shell-generated name.

See also

Open command

NewFolder — create a directory

Syntax

NewFolder *name*...

Description

Creates new directories with the names specified. Any parent directories included in the *name* specification must already exist.

Note: This command can only be used on hierarchical file system (HFS) disks.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned to the Shell:

0	Folders were created for each name listed
1	Syntax error
2	An error occurred
3	Attempt to use NewFolder on non-HFS volume

Examples

```
Newfolder Memos
```

Create Memos as a subdirectory of the current directory.

```
Newfolder Parent :Parent:Kid
```

Create Parent as a subdirectory of the current directory, and Kid as a subdirectory of Parent.

Open — open a window

Syntax

Open [-n | -r] [-t] [*name*]

Description

Opens a file as the active (topmost) window. If neither *name* nor the **-n** option is specified, an error results. If *name* is already open as a window, that window becomes the active (topmost) window.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Open returns the following status values:

0	No errors
1	Error in parameters
2	Unable to complete operation
3	System error

Options

-n	A new window is opened with the title *name*. If *name* is not specified, a unique name is generated for the new window. If file *name* already exists, that file is opened.
-r	Opens a read-only window associated with the file *name*. If file *name* doesn't exist, an error occurs.
-t	Open the window as the target window rather than as the active window (that is, make it the second window from the top). This option is identical to the Target command.

Examples

```
Open Test.a
```

Open the window Test.a.

```
Open -n
```

Open a new window with a Shell-generated name, Untitled-*n*.

See also Target, New, and Close commands

Parameters — write parameters

Syntax

Parameters [*parameters* ...]

Description

The Parameters command writes its parameters, including its name, to standard output. The parameters are written one per line, and each is preceded by its parameter number (in braces) and a blank. This command is useful for checking the results of variable substitution, command substitution, quoting, blank interpretation, and filename generation.

Input

None.

Output

Parameters are written to standard output.

Diagnostics

None.

Status

A status value of 0 is always returned.

Examples

```
Parameters One Two "and Three"
```

Writes the following three lines to standard output:

```
{0} Parameters
{1} One
{2} Two
{3} and Three
```

Recall that "..." and '...' quotes are removed before parameters are passed to commands.

See also

Echo command
"Parameters to Command Files" in Chapter 3

Pascal — Pascal Compiler

Syntax

Pascal [*option*...] [*file*...]

Description

Compiles the specified Pascal source files (programs or units). You can specify zero or more filenames. Each file is compiled separately—compiling file *Name*.p creates object file *Name*.p.o. By convention, Pascal source filenames end in a ".p" suffix.

See the manual *MPW Pascal Reference* for details of the language definition.

Input

If no filenames are specified, standard input is compiled, with output directed to the file p.o. You can terminate input by typing Command-Enter.

Output

Nothing is written to standard output. For each input file *name*, object code is sent to the file *name*.o.

Diagnostics

Errors are written to diagnostic output. Progress and summary information is also written to diagnostic output if the **-p** option is selected.

Status

The following status values are returned to the Shell:

0	Successful completion
1	Error in parameters
2	Compilation halted

Options

-b Generate A5-relative references whenever the address of a procedure or function is taken. (By default, PC-relative references are generated for routines in the same segment.)

-c Syntax check only—no object file is generated.

-d *name*=TRUE | FALSE
Set the compile time variable *name* to TRUE or FALSE.

-e *errLogFile* Write all errors to the error log file *errLogFile*. A copy of the error report will still be sent to diagnostic output.

-g Go directly to code generation, skipping the compilation pass. For an input filename Foo.p, an intermediate file Foo.p.o.i is expected as input to the process. Such a file is created if the Compiler crashes during code generation (for example, if the disk was full), or was aborted.

Note: The "beachball" cursor spins clockwise during compilation and counter-clockwise during code generation.

-i *pathname*[,*pathname*]...

Search for include or USES files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is as follows:

1. In the case of a USES filename, if no prior $U filename was specified, the filename is assumed to be the same as the unit name (with a ".p" appended).
2. The filename is used as specified. If a *full pathname* is given, then no other searching is applied.

   If the file wasn't found, and the pathname used to specify the file was a *partial pathname* (no colons in the name or a leading colon), then the following directories are searched.
3. The directory containing the current input file.
4. The directories specified in **-i** options, in the order listed.
5. The directories specified in the Shell variable {PInterfaces}.

The source filenames specified on the command line must include any relevant prefixes.

-k *prefixpath* Put the files specified in $LOAD commands in the directory specified by *prefixpath*.

-o *objName* Specify the pathname for the generated object file. If *objName* ends with a colon (:), it indicates a directory for the output file, whose name is then formed by the normal rules (that is, *inputFilename*.o). If the source file name contains a pathname, it is stripped off before *objName* is used as a prefix. If *objName* does not end with a colon, the object file is written to the file *objName*. (In this case, only one source file should be specified.)

-opt Suppress register code optimizations.

-ov Turn on overflow checking. (Warning: This may significantly increase code size.)

-p	Supply progress and summary information to diagnostic output, including Compiler header information (copyright notice and version number), module names and code sizes in bytes, and number of errors and compilation time.
-r	Suppress range checking.
-s	Enable swapping—the Compiler runs much more slowly, but uses less memory.
-t	Report compilation time to diagnostic output. The **-p** option also reports the compilation time.
-y *pathname*	Put the Compiler's temporary intermediate ("‧o.i") files in the directory specified by ***pathname***. (See also the **-g** option.)
-z	Turn off the output of embedded procedure names in the object code. This option is equivalent to specifying {$D-} in the source code.

Examples

```
Pascal Sample.p
```

Compile the Sample program provided in the PExamples folder.

```
Pascal File1.p File2.p -r
```

Compile File1.p and File2.p, producing object files File1.p.o and File2.p.o, and performing no range checking.

Note

Listing files are not produced directly by the Compiler. Refer to the PasMat and PasRef tools.

Availability

The Pascal Compiler is available as part of a separate Apple product, MPW Pascal.

See also

PasMat and PasRef commands

PasMat — Pascal program formatter ("pretty-printer")

Syntax

PasMat [*option*...] [*inputfile* [*outputfile*]]

Description

Reformats Pascal source code into a standard format, suitable for printouts or compilation. PasMat accepts full programs, external procedures, blocks, and groups of statements.

Note: A syntactically incorrect program causes PasMat to abort. If this happens, the generated output will contain the formatted source up to the point of the error.

PasMat options let you do the following:

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

PasMat specifications can be made through PasMat options or through special formatter directives, which resemble Pascal Compiler directives, and are inserted into the source file as Pascal comments. PasMat's default formatting is straightforward, and does not necessarily require you to use any options. The best way to find out how PasMat formats something is to try out a small example and see.

See Appendix K of the manual *MPW Pascal Reference* for details of PasMat directives and their functions.

Input

If no input files are specified, standard input is formatted.

Output

If no output file is specified, the formatted output is written to standard output. Refer to "Error Handling" below for more information about PasMat's treatment of errors in the source.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

Most of the following options modify the initial default settings of the directives described in Appendix K of the *MPW Pascal Reference manual.*

-a	Set **a-** to disable CASE label bunching.
-b	Set **b+** to enable IF bunching.
-body	Set **body+** to align procedure bodies with their enclosing BEGIN/END pair.
-c	Set **c+** for placement of BEGIN on same line as previous word.
-d	Set **d+** to enable the replacement of (* *) with { } comment delimiters.
-e	Set **e+** to capitalize identifiers.
-entab	Replace runs of blanks with tabs. The tab value is determined by the **-t** option or current **t=*n*** directive (*not* by the file's tab setting).
-f	Set **f-** to disable formatting.
-g	Set **g+** to group assignment and call statements.
-h	Set **h-** to disable FOR, WHILE, and WITH bunching.
-i *pathname*[,*pathname*]...	Search for include files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order for includes is specified under the description of the **-i** option for the Pascal command. (Note however that USES are not processed by PasMat.)
-in	Set **in+** to process Pascal compiler includes. This option is implied if the **-i** option is used.
-k	Set **k+** to indent statements between BEGIN/END pairs.
-l	Set **l+** for literal copy of reserved words and identifiers.

-list *listingFile*	Generate a listing of the formatted source. The listing is written to the specified file.
-n	Set **n+** to group formal parameters.
-o *width*	Set the output line width. The maximum value allowed is 150. The default is 80.
-p	Display version information and progress information to diagnostic output.
-pattern *=pattern=replacement=*	Process includes (**-in**) and generate a set of output files with exactly the same include structure as the input, but with new names. The new output filenames and include directives are generated by editing the input (or include) filenames according to the *pattern* and *replacement* strings. *Pattern* is a pathname to be looked for in the input file and in each include file (the *entire* pathname is used, and case is ignored). If the pattern is found, it is replaced by the *replacement* string. The result is a new pathname, which becomes the name for an output file. For example, `PasMat -pattern =OldFile=NewFile=` replaces each name containing the string "`OldFile`" with the string "NewFile". *Note:* Any character not contained in the *pattern* or *replacement* strings can be used in place of an equal sign. Special characters must be quoted. (See "Examples" below.)
-q	Set **q+** not to treat the ELSE IF sequence specially.
-r	Set **r+** to uppercase reserved words.
-rec	Indent a RECORD's field list under the record identifier.

-s *renameFile*	Rename identifiers. *RenameFile* is a file containing a list of identifiers and their new names. Each line in this file contains two identifiers of up to 63 characters each: The first name is the identifier to be renamed; the second name will replace all occurrences of the first identifier in the output. There must be at least one space or tab between the two identifiers. Leading and trailing spaces and tabs are optional. The case of the first identifier doesn't matter, but the second identifier must be specified exactly as it is to appear in the output. The case of all identifiers not specified in the ***renameFile*** is subject to the other case options (**-e**, **-l**, **-u**, and **-w**) or their corresponding directives. Reserved words cannot be renamed.
-t *tab*	Set the tab amount for each indentation level. If the **-entab** option is also specified, tab characters will actually be generated. The default tab value is 2.
-u	Rename all identifiers based on their first occurrence in the source. Specifications in the rename (**-s**) file always have precedence over this option—that is, the identifier's translation is based on the rename file rather than on the first occurrence .
-v	Set **v+** to put THEN on a separate line.
-w	Set **w+** to uppercase identifiers.
-x	Set **x+** to suppress space around operators.
-y	Set **y+** to suppress space around :=.
-z	Set **z+** to suppress space after commas.
-:	Set **:+** to align colons in VAR declarations (only if a **j** PasMat directive in the source specifies a *width*).
-@	Set **@+** to force multiple CASE tags onto separate lines.
"-#"	Set **#+** for "smart" grouping of assignment and call statements (grouped assignment and call statements on an input line will appear grouped on output). *Note:* Because **#** is the Shell's comment character, this option must be quoted on the command line.
-_	Set _+ for "portability" mode (underscores are deleted from identifiers).

All options except for **-list**, **-pattern**, **-s**, and **-entab** have directive counterparts. It's recommended that you specify the options as directives in the input source so that you won't have to specify them each time you call PasMat.

{PasMatOpts} variable. You can also specify a set of default options in the exported Shell variable {PasMatOpts}—PasMat processes these options before it processes the command-line options. {PasMatOpts} should contain a string (maximum length 255) specifying the options exactly as you would specify them on the command line (*except* for command-line quoting, which should be omitted; also note that the options **-pattern**, **-list** **-s**, and **-i**, which require a string parameter, can only be specified on the command line). For example, you can define {PasMatOpts} to the Shell (perhaps in the UserStartup file) as follows:

```
Set PasMatOpts "-n -u -r -d -entab -# -o 82 -t 2"
Export PasMatOpts
```

The entire definition string must be quoted to preserve the spaces.

As an alternative to specifying the options directly, you can indicate that the options are stored in a file, by specifying the file's full pathname prefixed with the character ^:

```
Set PasMatOpts "^pathname"
Export PasMatOpts
```

PasMat will now look for the default options in the specified file. The lines in this file can contain any sequence of command-line options (*except* for **-pattern**, **-list**, **-s**, and -I), grouped together on the same or separate lines. The lines may be commented by placing the comment in braces ({...}). A typical options file might appear as follows:

```
-n      {group formal params on same line}
-u      {auto translation of id's based on 1st occurrence}
-r      {uppercase reserved words}
-d      {leave comment braces alone}
-entab  {place real tabs in the output}
-#      {smart grouping}
-o 82   {output line width}
-t 2    {indent tab value}
```

(Except for the tab value, this example shows the recommended set of options.)

If PasMat does find a default set of options, then those options will be echoed as part of the status information given with the **-p** option.

Example

```
Pasmat -n -u -r -d -pattern "==formatted/="  Sample.p ∂
  "formatted/Sample.p"
```

Format the file Sample.p with the **-n**, **-u**, **-r**, and **-d** options, and write the output to the file "formatted/Sample.p". Includes are processed (**-pattern**) and each Pascal Compiler $I include file causes additional output files to be generated. Each of these files is created with the name "formatted/*filename*", where *filename* is the filename specified in the corresponding include. (The **-pattern** parameter contained a null pattern (==) with "formatted/" as a replacement string—a null pattern always matches the start of a string.)

Care must be taken when a command line contains quotes, slashes, or other special characters that are processed by the Shell itself. In this example, we used the slash character, so the strings containing it had to be quoted.

Limitations

PasMat has the following limitations:

- The maximum length of an input line is 255 characters.
- The maximum output line length is 150 characters.
- The input files and output files must be different.
- Only syntactically correct programs, units, blocks, procedures, and statements are formatted. This limitation must be taken into consideration when separate include files and conditional compiler directives are to be formatted.
- The Pascal include directive should be the last thing on the input line if includes are to be processed. Includes are processed to a maximum nesting depth of five. All includes not processed are summarized at the end of formatting. (This assumes, of course, that the **in** directive/option is in effect.)
- The identifiers CYCLE and LEAVE are treated as reserved Pascal keywords by PasMat. They are treated as two loop control statements by Pascal unless explicitly declared.

Error handling. The following errors are detected and written to diagnostic output:

- In general, premature end-of-file conditions in the input are not reported as errors, to accommodate formatting of individual include files, which may be only program segments. There are cases, however, where the include file is a partial program, which PasMat interprets and reports as a syntax error.
- There is a limit on the number of indentation levels that PasMat can handle. If this limit is exceeded, processing will abort. This problem should be exceedingly rare.
- If a comment would require more than the maximum output length (150) to meet the rules given, processing will abort. This problem should be even rarer than indentation level problems.

If a syntax error in the input code causes formatting to abort, an error message will give the input line number on which the error was detected. The error checking is not perfect—successful formatting is no guarantee that the program will compile.

Availability

PasMat is available as part of a separate Apple product, MPW Pascal.

See also Appendix K of the *MPW Pascal Reference* manual
Pascal and PasRef commands

PasRef — Pascal cross-referencer

Syntax

PasRef [*option* ...] [*sourceFile* ...]

Description

Reads Pascal source files, and writes a listing of the source followed by a cross-reference listing of all identifiers. Each identifier is listed in alphabetical order, followed by the line numbers on which it appears. Line numbers can refer to the entire source file, or can be relative to individual include files and units. Each reference indicates whether the identifier is defined, assigned, or simply named (for example, used in an expression).

See the manual *MPW Pascal Reference* for details of the Pascal language.

Identifiers may be up to 63 characters long, and are displayed in their entirety unless overridden with the **-x** directive. Identifiers may remain as they appear in the input, or they can be converted to all lowercase (**-l**) or all uppercase (**-u**).

For include files, line numbers are relative to the start of the include file; an additional key number indicates which include file is referred to. A list of each include file processed and its associated key number is displayed prior to the cross-reference listing.

USES declarations can also be processed by PasRef (their associated $U *filename* compiler directives are processed as in the Pascal Compiler). These declarations are treated exactly like includes, and, as with the Compiler, only the outermost USES declaration is processed (that is, a used unit's USES declaration is not processed).

As an alternative to processing USES declarations, PasRef accepts multiple source files. Thus you cross-reference a set of main programs together with the units they use. All the sources are treated like include files for display purposes. In addition, PasRef checks to see whether it has already processed a file (for example, if it appeared twice on the input list, or if one of the files already used or included it). If it has already been processed, then the file is skipped.

Input

If no filenames are specified, standard input is processed. Unless the **-d** option is specified, multiple source files are cross-referenced as a whole, producing a single source listing and a single cross-reference listing. Specifying the **-d** option is the same as executing PasRef individually for each file.

Output

All listings are written to standard output. Form feed characters are placed in the file before each new source listing and its associated cross reference. Pascal $P (page eject) compiler directives are also processed by PasRef, which may generate additional form feeds in the standard output listing.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

-a Process all files even if they are duplicates of ones already processed. The default is to process each (include) file or used unit only once.

-c Do *not* process a unit if the unit's filename is specified in the list of files to be processed on the command line, or if the unit has already been processed.

-d Treat each file specified on the command line as distinct. The default is to treat the entire list of files as a whole, producing a single source listing and a single cross-reference listing. This option is the same as executing PasRef individually for each specified file.

-i *pathname* [,*pathname*]...
Search for include or USES files in the specified directories. Multiple **-i** options may be specified. At most 15 directories will be searched. The search order is specified under the description of the **-i** option for the Pascal command.

-l Display all identifiers in the cross-reference table in lowercase. This option should not be used if **-u** is specified, but if it is, the **-u** is ignored.

-ni | **-noincludes**
Do not process include files. The default is to process the include files.

-nl | **-nolisting**
Do not display the input source as it is being processed. (The default is to list the input.)

-nolex	Do not display the lexical information on the source listing. See the "Examples" section for further details.
-nt | **-nototal**	Do not display the total line count in the source listing. This option is ignored if no listing is being generated (**-nl**).
-n[u] | **-nouses**	Do not process USES declarations. (The default is to process USES declarations.) If **-nu** is specified, then the **-c** option is ignored.
-o	The source file is an Object Pascal program. The identifier OBJECT is considered as a reserved word so that Object Pascal declarations may be processed. The default is to assume the source is not an Object Pascal program.
-p	Write version and progress information to diagnostic output.
-s	Do not display include and USES information in the listing or cross reference, and cross-reference by total source line number count rather than by include-file line number.
-t	Cross-reference by total source line-number count rather than by include-file line number. This option can be used if you are not interested in knowing the positions in included files. However, the include information is still displayed (unless **-s**, **-nl**, or **-nu** is specified). This option is implied by the **-s** option.
-u	Display all identifiers in the cross-reference table in uppercase. This option should not be used if **-l** is specified.
-w *width*	Set the maximum output width of the cross-reference listing. This setting determines how many line numbers are displayed on one line of the cross-reference listing. It does not affect the source listing. *Width* can be a value from 40 to 255; the default is 110.
-x *width*	Set the maximum display width for identifiers in the cross-reference listing. (The default is to set the width to the size of the largest identifier cross-referenced.) If an identifier is too long to fit in the specified width, it is indicated by preceding the last four characters with an ellipsis (...). *Width* can be a value from 8 to 63.

Normally both include files and USES declarations are processed. The **-noincludes** option suppresses processing of includes. The **-nouses** option suppresses processing of USES.

Omitting the **-nouses** option causes PasRef to process a USES declaration exactly as does the Pascal Compiler. However, you may want to cross-reference an entire system, including all of the units of that system. Processing the units through the USES declaration would cause only the INTERFACE section of each unit to be processed. If you use the **-nouses** option, then USES will not be processed and each unit from the list can be cross-referenced, treating the entire list as a single source.

PasRef can also cross-reference all the units of a program while still expanding other units not directly part of that program, such as the Toolbox units. In that case the **-c** option should be used. With the **-c** option, if the ($U interface) filename is the same as one of the filenames specified on the list, then the unit will not be processed from the USES declaration, because its full source will be (or has been) processed.

To summarize, you have the following choices:

- Don't process the USES, and specify a list of all files you want to process, by using the **-nouses** option.
- Process only the INTERFACEs through the USES declarations (like the Compiler), by omitting the **-nouses** option.
- Process some of the units through the USES and others as full sources, by specifying the **-c** option.

In all cases where a list of files is specified, no unit will ever be processed more than *once* (unless the **-a** option is given).

Example

```
PasRef -nu -w 80  Memory.p > Memory.p.Xref
```

Cross-reference the sample desk accessory Memory.p and write the output to the file Memory.p.Xref. No USES are processed (**-nu**). The following source and cross-reference listings are generated:

```
 1  1    1 --    {-----------------------------------------------------------
 2  1    2 --    Memory.p -- reports the amount of free space in the application
 3  1    3 --           and system heap
 4  1    4 --    -----------------------------------------------------------}
 5  1    5 --
 6  1    6 --    UNIT Memory;
 7  1    7 --
 8  1    8 --    INTERFACE
 9  1    9 --
10  1   10 --    USES
11  1   11 --                  MemTypes,
12  1   12 --                  QuickDraw,
13  1   13 --                  OSIntf,
14  1   14 --                  ToolIntf,
15  1   15 --                  PackIntf;
16  1   16 --
17  1   17 --    {$D+} { procedure names in the code, please }
```

```
 18   1   18 --      {$R-} { If we make a mistake, make it big }
 19   1   19 --
 20   1   20 --      { These 5 routines must be declared as below. Calls to your DA
 21   1   21 --      are dispatched to the appropriate routine by DRVRRuntime }
 22   1   22 --
 23   1   23 --      FUNCTION DRVROpen    (ctlPB: ParmBlkPtr;
 24   1   24 --                            dCtl:  DCtlPtr):     OSErr;
```
et cetera
```
222   1  222 --      END. {of memory UNIT}
223   1  223 --
```

Each line of the source listing is preceded by five columns of information:

1: The total line count.

2: The include key assigned by PasRef for an include or USES file. (See below.)

3: The line number within the include or main file.

4: Two indicators (left and right) that reflect the static block nesting level. The left indicator is incremented (mod 10) and displayed whenever a BEGIN, REPEAT, or CASE is encountered. On termination of these structures with an END or UNTIL, the right indicator is displayed, then decremented. It is thus easy to match BEGIN, REPEAT, and CASE statements with their matching terminations.

5: A letter that reflects the static level of procedures. The character is updated for each procedure nest level ("A" for level 1, "B" for level 2, and so on), and displayed on the line containing the heading, and on the BEGIN and END associated with the procedure body. Using this column you can easily find the procedure body for a procedure heading when there are nested procedures declared between the heading and its body.

The cross-reference listing follows:

```
 1. memory.p

-A-
  accEvent        146*( 1)  167 ( 1)
  accRun          147*( 1)  182 ( 1)
  ApplicZone      125 ( 1)
  Away            153*( 1)  168 ( 1)

-B-
  BeginUpdate     175 ( 1)
  Bold             93 ( 1)  121 ( 1)
  Boolean         151*( 1)

-C-
  csCode          165 ( 1)
  CSParam         168 ( 1)
  ctlPB            47*( 1)   50*( 1)   53*( 1)   56*( 1)   59*( 1)  143*( 1)
                  165 ( 1)  168 ( 1)  193*( 1)  219*( 1)  231*( 1)  238*( 1)
```

```
-D-
 dCtl             48*( 1)   51*( 1)   54*( 1)   57*( 1)   60*( 1)  144*( 1)
                 194*( 1)  202 ( 1)  209 ( 1)  211 ( 1)  220*( 1)  223 ( 1)
                 225 ( 1)  226 ( 1)  232*( 1)  239*( 1)
 DCtlPtr          48 ( 1)   51 ( 1)   54 ( 1)   57 ( 1)   60 ( 1)  144 ( 1)
                 194 ( 1)  220 ( 1)  232 ( 1)  239 ( 1)
 dCtlRefNum      209 ( 1)
 dCtlWindow      202 ( 1)  211=( 1)  225 ( 1)  226=( 1)
 dCtlwindow      223 ( 1)
 DisposeWindow   225 ( 1)
 DrawString       92 ( 1)  123 ( 1)  128 ( 1)  134 ( 1)  137 ( 1)  139 ( 1)
```
et cetera

```
*** End PasRef: 100 id's   230 references
```

The numbers in parentheses following the line numbers are the include keys of the associated include files (shown in column 2 of the source listing). The include file names are shown following the source listing. Thus you can see what line number was in which include file. An asterisk (*) following a line number indicates a definition of the variable. An equal sign (=) indicates an assignment. A line number with nothing following it means a reference to the identifier.

Limitations

PasRef does not process conditional compilation directives! Thus, given the "right" combination of $IFC's and $ELSEC's, PasRef's lexical (nesting) information can be thrown off. If this happens, or if you just don't want the lexical information, you may specify the **-nolex** option.

PasRef stores all its information on the Pascal heap. Up to 5000 identifiers can be handled, but more identifiers will mean less cross-reference space. A message is given if PasRef runs out of heap space.

Note: Although PasRef never misses a reference, it can infrequently be fooled into thinking that a variable is defined when it actually isn't. One case where this happens is in record structure variants. The record variant's case tag is always flagged as a definition (even when there is no tag type) and the variant's case label constants (if they are identifiers) are also sometimes incorrectly flagged depending on the context. (This occurs only in the declaration parts of the program.)

The identifiers CYCLE and LEAVE are treated as reserved Pascal keywords by PasRef. These are treated as two loop control statements by Pascal unless explicitly declared.

Availability

PasRef is available as part of a separate Apple product, MPW Pascal.

See also

MPW Pascal Reference

Pascal command

Paste — replace selection with Clipboard contents

Syntax

Paste [-c *count*] *selection* [*window*]

Description

Finds *selection* in the specified window and replaces its contents with the contents of the Clipboard. If no window is specified, the command operates on the target window (the second window from the top). It's an error to specify a window that doesn't exist.

For a definition of *selection*, see "Selections" in Chapter 4; a summary of the selection syntax is contained in Appendix B.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	At least one instance of the selection was found
1	System error
2	Any other error

Options

-c *count*	For a count of *n*, replace the next *n* instances of the selection with the contents of the Clipboard.

Examples

```
Paste §
```

Replace the current selection with the contents of the Clipboard. This command is like the Paste item in the Edit menu, except that the action occurs in the target window.

```
Paste /BEGIN/:/END/
```

Select everything from the next BEGIN to the following END, and replace the selection with the contents of the Clipboard.

See also Copy, Cut, and Replace commands
"Selections" in Chapter 4

Print — print text files

Syntax

Print [*option*...] [*file*...]

Description

Prints text files on the currently selected printer. (Printers are selected with the Chooser desk accessory.) One or more files may be printed.

Note: Print does not support font substitution. To print in a font other than that indicated in the resource fork of the file, use the **-font** option.

Important: Print requires the printer drivers available on version 1.0 (or later) of the *Printer Installation* disk.

Input

If no files are specified, Print reads from standard input. You can terminate input by typing Command-Enter.

Output

All output goes to the currently selected printer. Print sends no output to standard output.

Diagnostics

Errors and warnings are written to diagnostic output. If the **-p** option is specified, progress and summary information is also written to diagnostic output.

Status

The following status values are returned:

0	Successful completion
1	Parameter or option error
2	Execution error

Options

Note: The Print options can also be applied to the Print Window/ Print Selection menu item, by including them in the exported Shell variable {PrintOptions}. {PrintOptions} is originally set to `'-h'` in the Startup file.

-b	Print a round-rect border around the printable area of the page. Headers, if specified with the **-h** option, are separated from the body text by an extra line.
-c[opies] *n*	Print *n* copies of the file or selection.

-f[ont] *name*	Print using the font identified by *name* (for example, Courier). The default is the font indicated in information in the resource fork of the file, if present, and otherwise Monaco 9. (See also the **-size** option.) *Note:* Printing with a font that is not directly supported by the printer is significantly slower than printing with a built-in font.
-from *n*	Print pages starting from page number *n*. The default is to start with thefirst page of the file.
-h	Print page headers at the top of each page. The header indicates the time of printing, the name of the file, and the page number.
-hf[ont] *name*	Specify the font to be used in headers (**-h** option). The default is the font used in the file.
-hs[ize] *n*	Specify the font size to be used in headers. The default is 10.
-l[ines] *n*	Print (at most) *n* lines per page. Line spacing is adjusted so that the full page is used. If both **-l** and **-ls** are specified, the **-l** option takes precedence.
-ls *n*	Set line spacing. A value of 1 indicates normal (single) spacing (the default), 2 indicates double-spacing, and so on.
-n	Turn on line numbering; numbers appear to the left of the printed text.
-nw *n*	Specify the width of the line number (**-n**) field in characters. (The default is value is 5.) Negative values for *n* cause the field to be zero-padded.
-p	Write progress information to diagnostic output, indicating which files are printing and the number of lines and pages printed.
-page *n*	Number the pages of the file beginning with *n*. (By default, pages are numbered starting with 1.)
-q *quality*	Set print quality on the ImageWriter. *quality* is one of the following strings: `high` `standard` `draft` *Note:* This option is ignored when printing on the LaserWriter.

-r	Output pages to the printer in reverse order. This option eliminates the need to reorder pages on the LaserWriter.
-s[ize] *n*	Print using the font size identified by *n*. The default is to use the font size indicated in the resource fork of the file, if present; otherwise, the default size is 9.
-t[abs] *n*	Expand tabs, using the indicated tab setting. If this option isn't specified, the tab setting is taken from the resource fork of the file, if present; otherwise, the tab setting is taken from the {Tab} variable.
-title *name*	If printing page headers (with **-h**), use *name* as the title. (The default is to use the filename.)
-to *n*	Print pages up to page *n*. (The default is to print to the last page of the file.)

The following options control the page margins. *n* is the margin width in inches.

-tm *n*	Top margin. (Default = 0 inches)
-bm *n*	Bottom margin. (Default = 0 inches)
-lm *n*	Left margin. (Default = 0.2778 inches, for 3-hole punched pages)
-rm *n*	Right margin. (Default = 0 inches)

Examples

```
Print -h -size 8 -ls 0.85 Startup UserStartup
```

Print the files Startup and UserStartup with page headers, using Monaco 8 and compressing the line spacing.

```
Print
```

Print all text subsequently entered (that is, until you indicate end-of-file by typing Command-Enter).

See also

"Print..." menu item in Chapter 2

Rename — rename files and directories

Syntax

Rename [-y | -n] *name newName*

Description

The file, folder, or disk specified by *name* is renamed *newName*. A dialog box requests a confirmation if the rename would overwrite an existing file or folder. The -y or -n options can be used in command files to avoid this interaction.

Note: You can't use the Rename command to change the directory a file is in. To do this, use the Move command.

Note also: Wildcard renames in the following form **will not work:**

```
Rename ≈.text ≈.p
```

This is because the Shell expands the filename patterns "≈.text" and "≈.p" before invoking the Rename command. In order to gain the desired effect, you would need to execute a command such as the one shown in the fourth example below.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	Successful rename
1	Syntax error
2	*Name* does not exist
3	An error occurred

Options

-y	Answer "yes" to any confirmation dialog that may occur, causing conflicting files or folders to be deleted.
-n	Answer "no" to any confirmation dialog that may occur, skipping the rename for files or folders that already exist.

Examples

```
Rename File1 File2
```

Change the name of File1to File2.

```
Rename HD:Programs:Prog.c  Prog.Backup.c
```

Change the name of Prog.c in the directory HD:Programs to Prog.Backup.c in the same directory.

```
Rename Untitled: Backup:
```

Change the name of the disk Untitled to Backup.

To perform a wildcard rename, you could execute the following set of commands:

```
For Name In ≈.text
   ( Evaluate "{Name}" =~ /(≈)®1.text/ ) > Dev:Null
   Rename "{®1}.text" "{®1}.p"
End
```

The Evaluate command is executed only for its side effect of permitting regular expression processing. (The expression operator =~ indicates that the right-hand side of the expression is a regular expression.) Thus, you can use the regular expression capture mechanism, (*regularExpr*)®*n*. Evaluate's output is tossed in the bit bucket (Dev:Null).

See also

Move command

Alias command (for giving alternate names to a command)

Replace — replace the selection

Syntax

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

Description

Finds *selection* in the specified window and replaces it with *replacement*. If no window is specified, the command operates on the target window (the second window from the top). It's an error to specify a window that doesn't exist. If a count is specified, the Replace command will be repeated *count* times.

For a definition of *selection*, see "Selections" in Chapter 4. A summary of the selection syntax is contained in Appendix B.

The *replacement* may contain references to parts of the selection by using the ® operator. The expression ®*n*, where *n* is a digit, is replaced with the string of characters that matches the regular expression tagged by ®*n* in the selection. (See "Tagging Regular Expressions With the ® Operator" in Chapter 4.)

All searches are by default case insensitive. To specify case-sensitive matching, set the {CaseSensitive} variable before executing the command. (You can do this by selecting the Case Sensitive item from the Find menu.)

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	At least one instance of the selection was found
1	Syntax error
2	Any other error

Options

-c *count*	Repeat the command *count* times. As a convenience, ∞ (Option-5) can be specified in place of a number. -c ∞ replaces all instances of the selection from the current selection to the end of the document (or to the start of the document, for a backward search).

Examples

```
Replace -c ∞  /myVar/ 'myVariable'  Prog.p
```

Replace every subsequent instance of the selection with the string in single quotes.

```
Replace -c 5  /•[ ∂t]+/  ''
```

Strip off all the spaces and tabs at the front of the next five lines in the file (replace with the null string). This action takes place in the target window.

```
Set HexNum [0-9A-F]+
Set Spaces [ ∂t]+
Replace -c ∞  /({HexNum})®1{Spaces}({HexNum})®2/  ®1∂n®2
```

Define two variables for use in the subsequent Replace command, and convert a file that contains two columns of hex digits (such as the icon list from ResEdit) into a single column of hex digits.

See also

Find and Clear commands

Chapter 4, "Advanced Editing"

Appendix B, "Selections and Regular Expressions"

Request — request text from a dialog

Syntax

Request [-d *default*] *message*

Description

Displays an editable text dialog with OK and Cancel buttons and the prompt *message*. If the OK button is selected, then all text that the user typed into the dialog box is written to standard output. The **-d** option lets you set a default response to the request.

Input

None.

Output

Text from the dialog is written to standard output.

Diagnostics

None.

Status

The Request command returns the following status values

0	The OK button was selected
1	Syntax errors
2	The Cancel button was selected

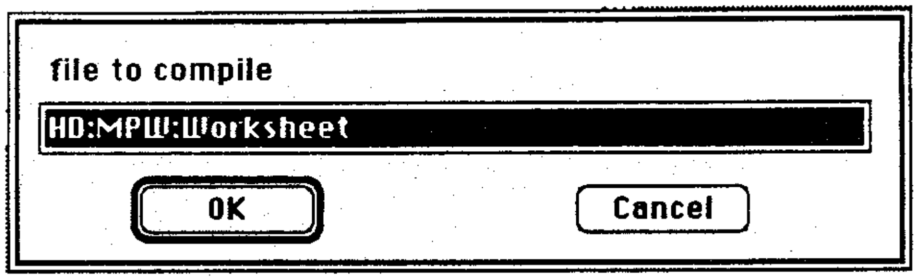
Options

-d *default*	The editable text field of the dialog is initialized to *default*. The default text appears in the dialog box—if the OK button is selected without changing the response, the default is written to standard output.

Examples

```
Set Exit 0
Set FileName "`Request 'File to compile' -d "{Active}"`"
If {FileName} ≠ ""
   Pascal "{FileName}" ≥≥ "{WorkSheet}"
End
Set Exit 1
```

Displays a dialog box that lets the user enter the name of a file to be compiled. Sets the default to be the name of the active window, as follows:


file to compile

HD:MPW:Worksheet

See also Alert and Confirm commands

Rez — Resource Compiler

Syntax

Rez [*option*...] [*resourceDescriptionFile*...]

Description

Creates the resource fork of a file according to a textual description. The resource description file is a text file that has the same format as the output produced by the Resource Decompiler, DeRez. The data used to build the resource file can come directly from the resource description file(s) as well as from other text files (via `#include` and `read` directives in the resource description file), and other resource files (via the `include` directive).

Rez includes macro processing, full expression evaluation, and built-in functions and system variables. For details of Rez, and the format of a resource description file, see Chapter 6. For a summary of the format of a resource description file, see Appendix D.

Input

Standard input is processed if no filenames are specified.

For all input files on the command line, the following search rules are applied:

1. Try to open the file with the name specified "as is."
2. If the preceding rule fails, and the filename contains no colons or begins with a colon, append the filename to each of the pathnames specified by the {RIncludes} variable and try to open the file.

Output

No output is sent to the standard output file. By default, the resource fork is written to the file **Rez.Out**. You can specify an output file with the **-o** option.

Diagnostics

If no errors or warnings are detected, Rez runs silently. Errors and warnings are written to diagnostic output.

Status

The following status codes are returned:

0	No errors
1	Error in parameters
2	Syntax error in file
3	I/O or program error

Options

-c[reator] *creatorExpr*
Set the output file creator. (The default value is '????'.)

-d[efine] *macro*[=*data*]
Define the macro variable *macro* to have the value *data*. If *data* is omitted, then *macro* is set to the null string—note that this still means that *macro* is defined. The **-d** option is the same as writing

```
#define macro [ data ]
```

at the beginning of the input.

-o *outputFile* Place the output in *outputFile*. The default output file is Rez.Out.

-p[rogress] Write version and progress information to diagnostic output.

-rd Suppress warning messages if a resource type is redeclared.

-ro Set the mapReadOnly flag in the resource map.

-t[ype] *typeExpr* Set the type of the output file. The default value is 'APPL'.

-u[ndef] *macro* Undefine the macro variable *macro*. This is the same as writing

```
#undef macro
```

at the beginning of the input. It is only meaningful to undefine the preset macro variables.

Example

```
Rez Types.r Sample.r  -o Sample
```

Generate a resource fork for the file Sample, based on the descriptions in Types.r and Sample.r.

See also

DeRez and RezDet commands

Chapter 6, "Using the Resource Compiler and Decompiler"

Standard resource type declarations in the {RIncludes} directory:

- Types.r
- SysTypes.r
- MPWTypes.r

Chapter 5, "Editing Resources With ResEdit"

RezDet — the resource detective

Syntax

RezDet [-b] [-q | -s | -d | -r | -l] *resourceFile*...

Description

If no options are specified, RezDet investigates the resource fork of each file for damage or inconsistencies. The specified files are read and checked one by one. Output is generated according to the options specified.

RezDet checks for the following conditions:

- The resource fork is at least the minimum size. (There must be enough bytes to read a resource header.)
- There is no overlap or space between the header, the resource data list, and the resource map. There should be no bytes between the EOF and the end of the resource map.
- Each record in the resource data list is used once and only once. The last data item ends exactly where the data list ends.
- Each item in the resource type list contains at least one reference; each sequence of referenced items starts where the previous resource type item's reference list ended; and each item in the reference list is pointed to by one and only one resource type list item.
- There are no duplicates in the resource type list.
- Each name in the name list has one and only one reference, and the last name doesn't point outside the name list.
- There are no duplicate names in the name list. Duplicate names cause an advisory warning rather than a true error. This warning is given only if the **-s**, **-d**, or **-r** option is selected.
- Each reference list item points to a valid data item and either has a name list offset of –1 or points to a valid name list offset.
- Bits 7 (Unused), 1 (Changed), or 0 (Unused) should not be set in the resource attributes.
- All names have a nonzero length.

Fields are displayed as hex or decimal for numeric values, or as a hex dump with associated printable Macintosh characters. The characters return ($0D), tab ($09), and null ($00) are displayed as "¬", "Δ", and "." respectively. The same is true for a resource name shown as text strings.

Note: RezDet does not use the Resource Manager and should not crash, no matter how corrupt the resource fork of the file.

Input

RezDet does not read from standard input.

Output

Information describing the resource fork is written to standard output (together with any other information generated by the **-s**, **-d**, **-l**, or **-r** options).

Diagnostics

Error messages go to diagnostic output.

Status

The following status values are possible:

0	No errors detected
1	Invalid options or no files specified
2	Resource format error detected
3	Fatal error—an I/O or program error was detected

Options

Only one of the following options can be used at one time:

-q[uiet] Don't write any information to standard output. This option suppresses all resource file format errors normally generated.

-s[how] Write information about each resource to standard output.

-d[ump] Same as **-show** but also generates detailed information about headers, name lists, data lists, and so on.

-r[awdump] Same as **-dump** but also dumps contents of data blocks, and so on.

Note: This option can generate ***huge*** amounts of output.

-l[ist] List resource types, IDs, names, attributes, and resource sizes to standard output in the following format:

'*type*' (*ID,name,attributes*) [*size*]

The following option can be used by itself or with other options:

-b[ig] Read the data for each resource into memory one resource at a time, instead of all at once (used for huge resource files). If RezDet tells you that it ran out of memory, try using this option.

Examples

```
RezDet "{SystemFolder}System"
```

Check the System file for damage.

```
RezDet -q Foo || Delete Foo
```

Remove the file Foo if the resource fork is damaged.

Limitations

Duplicate resource name warnings are generated even if the names belong to resources of different types.

The file attributes field in the resource map header is not validated.

The Finder-specific fields in the header and resource map header are ignored.

Save — save windows

Syntax

Save [-a | *window*]

Description

Saves the contents of *window* to disk, without closing *window*. The -a option saves all windows. Save with no parameters saves the target window (the second window from the top).

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Save returns the following status values:

0	No errors
1	Syntax error
2	Any other error

Options

-a	Save all open windows.

Examples

```
Save Test.c
```

Save the contents of the window titled Test.c.

```
Save "{Active}"
```

Save the contents of the active window.

```
Save "{Worksheet}"
```

Save the Worksheet window. This command is included in the Suspend and Quit files—it saves the Worksheet whenever you run an application or quit from the Shell.

Search — search files for a pattern

Syntax

Search [-l] *pattern* [*file...*]

Description

Searches the input files for lines that contain a pattern, and echoes them to standard output. If no input file is given, standard input is searched.

Pattern is a regular expression, optionally enclosed in forward slashes (/). *Pattern* is defined in "Pattern Matching" in Chapter 4 and in Appendix B.

Note: Pattern matching is by default case insensitive. To specify case-sensitive matching, set the {CaseSensitive} variable to a nonzero value and export it before executing the command. (You can do this by selecting the Case Sensitive item from the Find menu.)

Input

Standard input is read if no files are specified.

Output

Each line that contains the pattern is written to standard output. If more than one input file is given, Search prepends the filename to each output line.

Diagnostics

Error messages are written to diagnostic output.

Status

Search returns the following status values:

0	No error
1	Parameter error
2	Pattern not found

Options

-l	Display line numbers with the matched line(s).

Examples

```
Search /procedure/ Sample.p
```

Search the file Sample.p for the pattern "procedure". All lines containing this pattern are written to standard output.

```
Search /Export / "{MPW}"Startup "{MPW}"UserStartup
```

List the Export commands in the Startup and UserStartup files.

```
Search /PROCEDURE [a-zA-Z0-9_ ]*;/ "{PInterfaces}"≈
```

Search for the procedures with no parameters in the Pascal interface files supplied with MPW Pascal. Because more than one input file is specified, a filename will precede each line in the output.

```
Search -l /typedef[ ∂t]+struct/ "{CIncludes}"≈
```

List all lines containing stucture `typedefs` in the include files supplied with MPW C. Both the filename and the line number will be listed with each line that matches the pattern.

See also

Find command

"Pattern Matching (Using Regular Expressions)" in Chapter 4

Set — define or write Shell variable

Syntax

Set [*name* [*value*]]

Description

Assigns the string *value* to the variable *name*. If *value* is omitted, Set writes the name and its current value to standard output. If both *name* and *value* are omitted, Set writes a list of all variables and their values to standard output. (This output is in the form of Set commands.)

Note: To make variable definitions available to enclosed command files and programs, you must use the Export command.

Input

None.

Output

If *value* or both *name* and *value* are omitted, variable names and their values are written to standard output.

Diagnostics

Error messages are written to diagnostic output.

Status

The following status values are returned:

0	No error
1	Parameter error

Examples

```
Set CIncludes "{MPW}CFiles:CIncludes:"
```

Redefine the variable CIncludes.

```
Set CIncludes
```

Display the new definition of CIncludes.

```
Set Commands ∂
    ":,{MPW}Tools:,{MPW}Applications:,{MPW}ShellScripts:"
```

Redefine the variable {Commands} to include the directory "{MPW}ShellScripts:". (See Chapter 3 for a complete list of predefined variables.)

```
Set > SavedVariables
    # ... other commands
Execute SavedVariables
```

Write the values of all variables to file SavedVariables. Because the output of Set is actually Set commands, the file can be executed later to restore the saved variable definitions. This technique is used in the Suspend and Resume scripts to save and restore variable definitions, as well as exports, aliases, and menus.

See also

Export and Unset commands

"Defining and Redefining Variables" in Chapter 3

"The Startup and UserStartup File" in Chapter 3

Setfile — set file attributes

Syntax	Setfile [*option...*] *file...*
Description	Sets attributes for one or more files. The options apply to all files listed.
Input	None.
Output	None.
Diagnostics	Error messages are written to diagnostic output.

Status

The following status values are returned:

0	The attributes for all files were set
1	Syntax error
2	An error occurred

Options

-c *creator* Set the file creator. *Creator* must be 4 characters; for example,

```
-c 'MPS '
```

-t *type* Set the file type. *Type* must be 4 characters; for example,

```
-t 'TEXT'
```

-d *date* Set the creation date. *Date* is a string in the form

"*mm/dd/yy* [*hh:mm* [:*ss*] [AM | PM]]"

representing month, day, year (0-99), hour (0-23), minute, and second. The string must be quoted if it contains a space. A period (.) indicates the current date and time.

-m *date* Set the modification date: same format as above. A period (.) indicates the current date and time.

-l *h,v* Set the icon location. *h* and *v* are positive integer values and represent the horizontal and vertical pixel offsets from the upper-left corner of the enclosing window.

-a *attributes* — Set the file attributes. *Attributes* is a string composed of the characters listed below. Attributes that aren't listed remain unchanged.

L	Locked
V	Invisible
B	Bundle
S	System
I	Inited
D	on Desktop

Uppercase letters set the attribute to 1, lowercase to 0. For example,

```
Setfile -a vB
```

clears the invisible bit and sets the bundle bit.

Note: These attributes are described in the "File Manager" chapter of ***Inside Macintosh***. Note that setting the locked bit doesn't prevent the file from being changed.

Examples

```
Setfile -c "MPS " -t MPST ResEqual
```

Set the creator and type for the MPW Pascal tool ResEqual.

```
Setfile Foo -m "2/15/86 2:25"
```

Set file Foo's modification date.

```
Setfile Foo Bar -m .
```

Set the modification date to the current date and time (.). This is useful, for instance, before running Make.

See also

Files command (The **-l** option displays file information.)

Shift — renumber command-file parameters

Syntax

Shift [*number*]

Description

Renames the command-file positional parameters {*number*+1}, {*number*+2}... to {1}, {2}, and so on. If *number* is not specified, the default value is 1. Parameter 0 (the command name) is not affected. The variables {Parameters}, {"Parameters"}, and {#} variables are also modified to reflect the new parameters.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

The following status values are returned:

0	Success
1	Error in the parameter

Examples

The following command file, "FontMany," sets the font information for a list of windows.

```
#  FontMany fontName fontSize [window...]
Set Exit 0
Set fontName "{1}"
Set fontSize "{2}"
Shift 2
For window in {"Parameters"}
   Font "{fontName}" "{fontSize}" "{window}"
End
```

The Shift command removes FontMany's *fontName* and *fontSize* parameters from {"Parameters"}, so that {"Parameters"} can be used in the For command. The new command file could be called as follows:

```
FontMany Monaco 9 `windows`
```

That is, use Monaco 9 to display all the open windows.

See also

"Parameters to Command Files" in Chapter 3

Tab — set a window's tab setting

Syntax

Tab *number* [*window*]

Description

Sets the tab setting of the file in *window* to *number* spaces. If no window is specified, the command operates on the target window (the second window from the top). It's an error to specify a window that doesn't exist.

Note: The Tab command (and the Tabs… menu item) modify the tab setting of an existing window. The {Tab} variable is used to initialize the tab setting of a new window, or as the default for files with no tab setting.

Input

None.

Output

None.

Diagnostics

Errors are written to diagnostic output.

Status

Tab returns the following status values:

0	No errors
1	Syntax error
2	An illegal tab count was specified

Examples

```
Tab 4
```

Set the tab value of the target window to represent 4 spaces.

See also

Entab command

Target — make a window the target window

Syntax

Target *name*

Description

Makes window *name* the target window for editing commands (that is, the second window from the top). If window *name* isn't already open, then file *name* is opened as the target window. If *name* doesn't exist, an error is returned.

Input

None.

Output

None.

Diagnostics

Error messages are written to diagnostic output.

Status

Target returns the following status values:

0	No errors
1	Error in parameters
2	Unable to complete operation
3	System error

Examples

```
Target Sample.a
```

Make the window Sample.a the target window.

See also

"Editing With the Command Language" in Chapter 2

TLACvt — convert Lisa TLA Assembler source

Syntax

TLACvt [*option...*] [*sourceFile...*]

Description

Converts the specified Lisa Workshop TLA Assembler source files to the syntax required by the MPW Assembler. If the input file name is ***name***, the converted output is written to ***name***.a. The following elements are converted:

- tokens within statements
- special tokens within macros
- directives

For the details of these conversions, see "TLA Conversion" in Appendix E of the companion volume ***MPW Assembler Reference.***

The case (upper/lower) of directive names in the output may be controlled by editing the file TLACvt.Directives. This file contains a list of all the MPW Assembler directives needed for conversion. The pathname to this file must be specified with the **-f** option.

Input

If no filenames are specified, standard input is converted.

Output

If input is from the standard input file, the converted output is written to standard output. If the input file name is ***name***, the converted output is written to ***name***.a. You can use the **-n**, **-prefix**, and **-suffix** options to modify the output file naming conventions.

Diagnostics

Parameter errors and progress information are written to diagnostic output.

Status

The following status codes are returned to the Shell:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

-d	Detab the input. All tabs are removed and replaced with spaces. The number of spaces is determined by the tab setting. (See the -t option below.)
-e	Detab the input (as done by the **-d** option) and entab the output as a function of the tab setting. (See the **-t** option below.)

-f *directivesFile*	The casing of directives is controlled by the file of directives specified by *directivesFile*. The file TLACvt.Directives is supplied for this purpose; you can edit it to change the capitalization. By default, all directives are converted to uppercase.
-m	Do *not* insert TLA-compatible mode-setting directives (BLANKS ON and STRING ASIS) into converted source.
-n	Do not add the ".a" extension to the input filename to produce the output filename. If you specify this option, you must also specify **-prefix** or **-suffix**.
-p	Writes TLACvt's version information and conversion status to diagnostic output.
-pre[fix] *string*	If the input file name is Name, the output filename is produced by prefixing *string* to the name, that is, "*string*Name.a". (The ".a" suffix may be suppressed by using the **-n** option or changed by using the **-suffix** option.)
-suf[fix] *string*	If the input file name is Name, the output filename is produced by appending *string* to the file name, that is, "Name*string*". The default suffix is ".a".
-t *tabSetting*	Set the output file's tab value to *tabSetting* (2 to 255). The default is to use the input file's tab setting (if there is one); otherwise a value of 8 is assumed. (8 is the default used by the Lisa Workshop's MacCom utility when transferring text files—it's assumed that MacCom was used to transfer the TLA files from the Lisa to the Macintosh.)
-u *c*	When TLACvt detects a name in the opcode field that is the same as an MPW directive, it appends the character *c* to make the name unique. (The default character is #.)

Example

```
TLACvt -t 8  TLAFile1.Text  TLAFile2.Text
```

Convert the Lisa TLA Assembler source files TLAFile1.Text and TLAFile2.Text to the MPW Assembler source files TLAFile1.Text.a and TLAFile2.Text.a. The **-t** option sets the tab setting for both input files to 8, and entabs the output files based on a tab setting of 8.

Limitations

Limitations are noted in the detailed description of TLA conversions in *MPW Assembler Reference*.

See also Appendix E, "TLA Conversion," in *Macintosh Workshop Assembler Reference*
CvtObj command

Unalias — remove aliases

Syntax

Unalias [*name*...]

Description

Removes any alias definition associated with the alias *name*. (It is not an error if no definition exists for *name*.)

Caution: If no names are specified, all aliases are removed.

The scope of the Unalias command is limited to the current command file; that is, aliases in enclosing command files are not affected. If you are writing a command file that is to be completely portable across various users' configurations of MPW, you should place the command

```
Unalias
```

at the beginning of your file to make sure no unwanted substitutions occur.

Input

None.

Output

None.

Diagnostics

None.

Status

A status value of 0 is always returned.

Example

```
Unalias File
```

Remove the alias "File". (This alias is defined in the Startup file.)

See also

Alias command
"Command Aliases" in Chapter 3

Unmount — unmount volumes

Syntax

Unmount *volume*...

Description

Unmounts the specified volumes. A volume name must end with a colon (:). If ***volume*** is a number without a colon, it's interpreted as a disk drive number. The volumes cannot be referenced again until remounted. If you unmount the current volume (the volume containing the current directory), then the boot volume becomes the current volume.

Input

None.

Output

None.

Diagnostics

Error messages are written to diagnostic output.

Status

The following status values are returned:

0	The volume was successfully unmounted
1	Syntax error
2	An error occurred

Examples

```
Unmount Memos:
```

Unmount the volume titled Memos.

```
Unmount 1 2
```

Unmount the volumes in drives 1 (the internal drive) and 2 (the external drive). (The command `Eject 1 2` would unmount ***and*** eject the volumes.)

See also

Eject and Mount commands

Unset — remove Shell variables

Syntax

Unset [*name*...]

Description

Removes any variable definition associated with *name*. (It's not an error if no definition exists for *name*.)

Caution: If no names are specified, all variable definitions are removed. This can have serious consequences. For example, the Shell uses the variable {Commands} to locate utilities and applications, and uses several other variables to set defaults. The Assembler and Compilers use variables to help locate include files. (For details, see "Variables Defined in the Startup File" in Chapter 3.)

The scope of the Unset command is limited to the current command file; that is, variables in enclosing command files are not affected.

Input

None.

Output

None.

Diagnostics

None.

Status

A status value of 0 is always returned.

Example

```
Unset CaseSensitive
```

Remove the variable definition for {CaseSensitive}. (This turns off case-sensitive searching for the editing commands.)

See also

Set and Export commands

"Defining and Redefining Variables" in Chapter 3

Volumes — list mounted volumes

Syntax

Volumes [-l] [-q] [*volume*...]

Description

For each volume named, Volumes writes its name and any other information requested to standard output. The output is sorted alphabetically. A volume name must end with a colon (:)—if *volume* is a number without a colon, it's interpreted as a disk drive number. If *volume* is not given, all mounted volumes are listed.

Input

None.

Output

Information about the specified volumes is written to standard output.

Diagnostics

Error messages are written to diagnostic output.

Status

The following status values are returned:

0	No errors
1	Syntax error
2	No such volume

Options

-l	List volumes in long format, giving volume name, drive (0 if off-line), capacity, free space, number of files, and number of directories.
-q	Don't quote volume names that contain special characters. (The default is to quote names that contain spaces or other special characters.)

Examples

```
Volumes -l
```

will write information such as:

```
Name       Drive    Size     Free      Files    Dirs
HD:            3    19171K   14242K      290      33
```

```
Files `Volumes 1`
```

List the files on the disk in drive 1 (the built-in floppy disk drive).

Windows — list windows

Syntax

Windows [-q]

Description

Writes the full pathname of each file currently in a window. The names are written to standard output, one per line, from backmost to frontmost.

Input

None.

Output

The list of open windows is written to standard output.

Diagnostics

None.

Status

Status value 0 is always returned.

Options

-q	Don't quote window names that contain special characters. (The default is to quote names that contain spaces or other special characters.)

Examples

```
Windows
```

List the pathnames of all open windows.

```
Print {PrintOptions} `Windows`
```

Print all of the open windows, using the options specified by the {PrintOptions} variable. This example uses command substitution: Because the Windows command appears in backquotes (`...`), its output supplies the parameters to the Print command.

```
Begin
     Echo For window in "`Windows`"
     Echo  'Open "{window}" | Set Status 0'
     Echo End
End > SavedWindows
```

Write a command script, in the file SavedWindows, that will reopen the current set of open windows. Note how Echo is used to create the script, and that the I/O redirection following End applies to all of the enclosed commands. Also note the use of quoting: The `` `Windows` `` command is executed immediately; but, because it's in single quotes, the `"{window}"` variable isn't expanded until the SavedWindows file is executed. This technique is used in the Suspend script to save the list of open windows.

Appendix A

Macintosh Workshop Files

This appendix lists all of the files provided with the Macintosh Programmer's Workshop, including Macintosh Workshop Pascal and Macintosh Workshop C. The first list is an inclusive list of all MPW files. (Volume names are shown in bold; directory names begin and end with a colon.) Subsequent lists show the recommended arrangements of these files for an HD-20 and for a set of 800K disks.

Distribution files

MPW1:

MPW Shell	MPW Shell
StartUp	Startup command file
StartUp.800K	Startup file for 800K disk system
StartUp.XL	Startup file for Macintosh XL
UserStartup	User-specific startup file
Suspend	Suspend command file
Resume	Resume command file
Quit	Quit command file
MPW.Help	Command syntax descriptions (for Help command)
:System Folder:	
Finder	Finder
System	System file (System 3.2 or later is required for MPW)

MPW2:

SysErrs.Err	Indexed error message file (used by the Shell and tools)
:RIncludes:	
Types.r	Common resource type definitions
MPWTypes.r	MPW-specific resource type definitions
SysTypes.r	System resource type definitions
:Tools:	
Compare	Text file comparison tool
Count	Line count, character count tool
Entab	Entabbing tool
Lib	Library construction tool
Line	Command file for selecting error line
Link	Linker
Make	Program builder
Print	Print tool
Rez	Resource Compiler
Search	Search tool (regular expression processor)

MPW3:

:More Tools:	
Canon	Canonical spelling tool
Canon.Dict	Dictionary file for Canon
CvtObj	Lisa Workshop object file conversion tool
DeRez	Resource Decompiler
DumpCode	Code formatting tool
DumpObj	Object file formatting tool
FileDiv	File division tool
MDSCvt	MDS Assembler conversion tool
MDSCvt.Directives	Directives file for MDSCvt
RezDet	Resource detective
TLACvt	TLA Assembler conversion tool
TLACvt.Directives	Directives file for TLACvt

MPW4:

Asm	68xxx Assembler
:Applications:	
ResEdit	Interactive resource editor

:Debuggers:

MacsBug	MacsBug debugger
MacsBug.XL	Macintosh XL version of MacsBug

MPW5:

:AExamples:

Instructions.a	Instructions for building assembly-language examples
MakeFile.a	Makefile for building assembly-language examples
Sample.a	Source for *Inside Macintosh* sample application
Sample.r	Resource definitions for Sample application
IntEnv.a	Integrated Environment interface used in Count tool
Signal.a	Signal-handler interface used in Count
Count.a	Source for MPW Count tool
Stubs.a	Dummy library routines (used to override library Routines not used by MPW tools)
Memory.a	Source for Memory desk accessory

:AIncludes:

ATalkEqu.a	AppleTalk equates
FixMath.a	Fixed-point mathematics routines
FSEqu.a	File system equates, including hierarchical file system (HFS)
FSPrivate.a	File system low-level equates
Graf3DEqu.a	Graf3D interface
HardwareEqu.a	Hardware equates
IntEnv.a	Integrated Environment (MPW tool) equates
ObjMacros.a	Macros for object-oriented programming
PackMacs.a	Package macros, including List Manager
PrEqu.a	Printing equates
Private.a	Low-level system equates
QuickEqu.a	Quickdraw equates
SANEMacs.a	SANE (Standard Apple Numerics Environment) macros
SCSIEqu.a	SCSI port equates and trap macros
Signal.a	Signal handler equates
SonyEqu.a	Low-level Sony disk driver equates
SysEqu.a	System equates, including HFS
SysErr.a	System errors
TimeEqu.a	Time Manager queue element and local variable structure
ToolEqu.a	Macintosh toolbox equates
Traps.a	Trap macro definitions for toolbox calls

:Libraries:

AppleTalk	AppleTalk resources
SERD	Serial Driver resources
DRVRRuntime.o	Driver runtime library
Interface.o	"Inside Macintosh" interface library
ObjLib.o	Object-oriented programming library
Runtime.o	Runtime library for assembly language and Pascal
ToolLibs.o	MPW tool library (spinning cursor, error manager)

MPW Pascal files

Pascal1:

Pascal	Pascal Compiler
PasMat	Pascal print formatter ("pretty-printer")
PasRef	Pascal cross-referencer

Pascal2:

:PExamples:

Instructions.p	Instructions for building example programs
MakeFile.p	Makefile for Sample program
Sample.p	Sample application
Sample.r	Resource description file for Sample.p
ResEqual.p	Sample MPW tool
Stubs.a	Dummy library routines (used to override library routines not used by MPW tools)
Memory.p	Sample MPW tool
Memory.r	Resource description file for Memory.p
ResEd68K.a	Routines for extending ResEdit
ResEd.p	Routines for extending ResEdit
ResXXXXEd.p	Sample resource editor

:PInterfaces:

AppleTalk.p	AppleTalk interface
CursorCtl.p	MPW cursor-control interface
ErrMgr.p	MPW error manager interface
FixMath.p	Interface for fixed-point mathematics routines
Graf3D.p	Graf3D interface
IntEnv.p	Integrated Environment (MPW tool) interface
MacPrint.p	Printing interface
MemTypes.p	Common types
OSIntf.p	Operating system interface
PackIntf.p	Packages interface
PasLibIntf.p	Pascal library interface
Quickdraw.p	Quickdraw interface
SANE.p	SANE numerics interface
SCSIIntf.p	SCSI manager interface
Signal.p	Signal-handling interface
ToolIntf.p	Macintosh toolbox interface

:PLibraries:

PasLib.o	Pascal language library, included built-ins and I/O
SANELib.o	SANE numerics library

MPW C files

C1:

C	C Compiler

C2:

CExamples:

Instructions.c	Instructions for building examples
MakeFile.c	Makefile for building examples
Sample.c	Source for Sample application
Sample.r	Resource specifications for Sample application
Count.c	Source for Count tool
Stubs.c	Dummy library routines (used to override library routines not used by MPW tools)
Memory.c	Source for Memory desk accessory
Memory.r	Resource specifications for Memory desk accessory

CIncludes:

AppleTalk.h	AppleTalk header file
Controls.h	Control Manager header file
CType.h	Character types header file
Desk.h	Desk Manager header file
Devices.h	Device Manager header file
Dialogs.h	Dialog Manager header file
DiskInit.h	Disk Initialization header file
Disks.h	Disk Driver header file
ErrNo.h	Standard C Library error numbers
Errors.h	Macintosh libraries error numbers
Events.h	Toolbox Event Manager header file
FCntl.h	File controls header file
Files.h	File Manager header file
FixMath.h	Fixed-point math header file
Fonts.h	Font Manager header file
Graf3D.h	Graf3D header file
IOCtl.h	I/O Control header file
Lists.h	List Manager header file
Math.h	Mathematical functions header file
Memory.h	Memory Manager header file
Menus.h	Menu Manager header file
OSEvents.h	Operating System Event Manager header file
OSUtils.h	Operating System Utilities header file
Packages.h	Packages header file
Printing.h	Print Manager header file
Quickdraw.h	Quickdraw header file
Resources.h	Resource Manager header file
Retrace.h	Vertical Retrace header file
SANE.h	SANE header file
Scrap.h	Scrap Manager header file
SCSI.h	SCSI Manager header file
SegLoad.h	Segment Loader header file
Serial.h	Serial Driver header file
SetJmp.h	Setjmp header file
Signal.h	Signal handler header file
Sound.h	Sound Driver header file
StdIO.h	Standard I/O header file
Strings.h	String conversion header file
TextEdit.h	TextEdit header file
Time.h	Time Manager header file
ToolUtils.h	Toolbox Utilities header file
Types.h	Common types header file
Values.h	Arithmetic values header file
VarArgs.h	Variable argument list header file

Windows.h	Window Manager header file
:CLibraries:	
CInterface.o	Macintosh interface library for C only
CRuntime.o	C runtime library: "Main" entry point, data initialization, Quickdraw data, A5, low-level I/O, built-in routines for C, signal handling
CSANELib.o	SANE numerics library
Math.o	Math functions library
StdCLib.o	Standard C Library

Hard Disk 20 configuration

:System Folder:
- Finder
- System

:MPW:
- MPW Shell
- StartUp
- UserStartup
- Suspend
- Resume
- Quit
- MPW.Help
- SysErrs.Err
- **:RIncludes:**
  - Types.r
  - SysTypes.r
  - MPWTypes.r

:Tools:

Asm
C
Canon
Canon.Dict
Compare
Count
CvtObj
DeRez
DumpCode
DumpObj
Entab
FileDiv
Lib
Line
Link
Make
MDSCvt
MDSCvt.Directives
Pascal
PasMat
PasRef
Print
Rez
RezDet
Search
TLACvt
TLACvt.Directives

:Applications:

ResEdit

:Debuggers:

MacsBug

:AExamples:

Instructions.a
Makefile.a
Sample.a
Sample.r
IntEnv.a
Signal.a
Count.a
Stubs.a
Memory.a

:AIncludes:

- ATalkEqu.a
- FixMath.a
- FSEqu.a
- FSPrivate.a
- Graf3DEqu.a
- HardwareEqu.a
- ObjMacros.a
- PackMacs.a
- PrEqu.a
- Private.a
- QuickEqu.a
- SANEMacs.a
- SCSIEqu.a
- SonyEqu.a
- SysEqu.a
- SysErr.a
- TimeEqu.a
- ToolEqu.a
- Traps.a

:Libraries:

- AppleTalk
- SERD
- Interface.o
- ObjLib.o
- Runtime.o
- ToolLibs.o
- DRVRRuntime.o

:PExamples:

- Instructions.p
- MakeFile.p
- Sample.p
- Sample.r
- ResEqual.p
- Stubs.a
- Memory.p
- Memory.r
- DRVRHead.a
- ResEd68K.a
- ResEd.p
- ResXXXXEd.p

:PInterfaces:
- AppleTalk.p
- CursorCtl.p
- ErrMgr.p
- FixMath.p
- Graf3D.p
- IntEnv.p
- MacPrint.p
- MemTypes.p
- OSIntf.p
- PackIntf.p
- PasLibIntf.p
- Quickdraw.p
- SANE.p
- SCSIIntf.p
- Signal.p
- ToolIntf.p

:PLibraries:
- PasLib.o
- SANELib.o

:CExamples:
- Instructions.c
- MakeFile.c
- Sample.c
- Sample.r
- Count.c
- Stubs.c
- Memory.c
- Memory.r

:CIncludes:

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

:CLibraries:

CInterface.o
CRuntime.o
CSANELib.o
Math.o
StdCLib.o

800K disk configuration

MPW:

MPW Shell
StartUp (renamed from Startup.800K)
UserStartup
Suspend
Resume
Quit
MPW.Help
SysErrs.Err

:Debuggers:

MacsBug

:RIncludes:

Types.r
SysTypes.r
MPWTypes.r

:Tools:

Line
Link
Make
Print
Rez
etc.

:Libraries:

- AppleTalk
- SERD
- Interface.o
- ObjLib.o
- Runtime.o
- ToolLibs.o
- DRVRRuntime.o

:System Folder:

- Finder
- System

Note: You can free up more disk space by removing the Finder and making the MPW Shell the startup application.

Asm:

Asm

:AExamples:

- Instructions.a
- MakeFile.a
- Sample.a
- Sample.r
- IntEnv.a
- Signal.a
- Count.a
- Stubs.a
- Memory.a

:AIncludes:

ATalkEqu.a
FixMath.a
FSEqu.a
FSPrivate.a
Graf3DEqu.a
HardwareEqu.a
ObjMacros.o
PackMacs.a
PrEqu.a
Private.a
QuickEqu.a
SANEMacs.a
SCSIEqu.a
SonyEqu.a
SysEqu.a
SysErr.a
TimeEqu.a
ToolEqu.a
Traps.a

Pascal:

Pascal

:PExamples:

Instructions.p
MakeFile.p
Sample.p
Sample.r
ResEqual.p
Stubs.a
Memory.p
Memory.r
DRVRHead.a
ResEd68K.a
ResEd.p
ResXXXXEd.p

:PInterfaces:

- AppleTalk.p
- CursorCtl.p
- ErrMgr.p
- FixMath.p
- Graf3D.p
- IntEnv.p
- MacPrint.p
- MemTypes.p
- OSIntf.p
- PackIntf.p
- PasLibIntf.p
- Quickdraw.p
- SANE.p
- SCSIIntf.p
- Signal.p
- ToolIntf.p

:PLibraries:

- PasLib.o
- SANELib.o

C:

C

:CExamples:

- Instructions.c
- MakeFile.c
- Sample.c
- Sample.r
- Count.c
- Stubs.c
- Memory.c
- Memory.r

:CIncludes:

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

```
:CLibraries:
    CInterface.o
    CRuntime.o
    CSANELib.o
    Math.o
    StdCLib.o
```

Appendix B

Selections and Regular Expressions

This appendix formally defines the syntax of selections and regular expressions as used in the Shell command language. It also lists the Option-key characters used in selections and regular expressions. For examples of their use, see Chapter 4, "Advanced Editing."

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
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
$regularExpr_1$ $regularExpr_2$	$regular\text{-}expr_1$ followed by $regular\text{-}expr_2$
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 ***regular-expr*** can be referred to as ®*digit*
literal	
'*string*'	each character in *string* is taken literally
"*string*"	each character in *string* is taken literally, except for ∂ substitutions
characterExpr	
character	character (unless it's listed as special following the 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
character-List	
]	"]" first in list represents itself
–	"–" first in list represents itself
character	character
characterList character	list of characters
character$_1$ – *character*$_2$	character range from *character*$_1$ to *character*$_2$ inclusive

❖ *Note:* The regular expression operators ?, ≈, [...], *, +, and «...» 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

The operators are listed below beginning with the highest-precedence operators.

( )

? ≈ * + [ ] « » ®

concatenation

• ∞

Option-key characters

The following Option-key characters are used in selections and regular expressions:

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
Δ	Option-J	position
®	Option-R	tag operator
«	Option-\	encloses number of repetitions
»	Shift-Option-\	encloses number of repetitions

Appendix C

MPW Character Reference

This appendix gives a brief summary of the special operators used in the Macintosh Programmer's Workshop. 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 3 and 4.

Table C-1
MPW operators

Shell characters:

Operator	**Meaning**
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 quotes"—literalize *chars*
"*chars*"	"Soft quotes"—literalize *chars* except for {…} (variable substitution), \`…\` (command substitution), and ∂ (escape)
/*chars*/	Regular expression quotes—literalize /*chars*/ except for {…}, \`…\`, and ∂
\chars\\	Regular expression quotes—literalize \chars\\ except for {…}, \`…\`, and ∂
{*variable*}	Substitute *variable*
\`*command*\`	Substitute output of *command*
< *filename*	Redirect standard input
> *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->)

Selections (editing commands):

§	Current selection (§ is Option-6)
n	Line number *n*
!*n*	*n* lines after §
¡*n*	*n* lines before § (¡ is Option-1)
•	Beginning of file (• is Option-8)
∞	End of file (∞ is Option-5)
Δ*selection*	Beginning of *selection* (Δ is Option-J)
*selection*Δ	End of *selection* (Δ is Option-J)
selection!*n*	*n* characters before *selection*
selection¡*n*	*n* characters after *selection* (¡ is Option-1)
regExpr/	*regExpr* after current selection
\\ *regExpr*\\	*regExpr* before current selection

selection$_1$:*selection*$_2$	*selection*$_1$, *selection*$_2$ and in between

Regular expressions (and filename generation):

char	Match *char*
∂*char*	Literal *char* (∂ is Option-D)
'*char*...'	Literal *chars*
?	Any character
≈	Zero or more chars (short for ?*) (≈ is Option-X)
[*characterList*]	Any character in *characterList*
[¬*characterList*]	Any character not in *characterList* (¬ is Option-L)
*regExpr**	*regExpr* zero or more times
regExpr+	*regExpr* one or more times
regExpr«*n*»	*regExpr* *n* times (« and » are Option-\ and Option-Shift-\)
regExpr«*n*,»	*regExpr* *n* or more times
regExpr«n_1,n_2»	*regExpr* n_1 to n_2 times
(*regExpr*)	*regExpr* (grouping)
(*regExpr*)®*n*	Tagged *regExpr*, where $0 \le n \le 9$ (® is Option-R)
regExpr$_1$*regExpr*$_2$	*regExpr*$_1$ followed by *regExpr*$_2$
regExpr	*regExpr* at beginning of line (• is Option-8)
regExpr∞	*regExpr* at end of line (∞ is Option-5)

Shell numbers:

$*nnn*	Hexadecimal number
0x*nnn*	Hexadecimal 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-=)
=~	Matches regular expression
!~	Does not match regular expression
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&& AND	Logical AND
|| OR	Logical OR

Appendix D

Resource Description Syntax

Syntax Notation xx

Structure of a Resource Description File xx
Include — include resources from another file x
Read — read data as a resource x
Data — specify raw data x
Type — declare resource type x
    Data-types
    Fill-type
    Alignment
    Switch-type
    Array-type
Resource — specify resource data x

Preprocessor Directives xx

Syntax xx
Identifiers x
Token Delimiters x
Compound Types x
Expressions x
Numbers and Literals x
Variables x
Strings x

This appendix defines the form of a resource description file, used by the Resource Compiler and Decompiler. For a full explanation, see Chapter 6, "Using the Resource Compiler and Decompiler."

Syntax Notation

The following syntax notation is used in this appendix:

`terminal`	Must be entered as shown.
non-terminal	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 quotes; for example,

{ '{' *data-string* '}' }?

—that is, a *data-string* is optional, and must be enclosed in braces, if included. Otherwise, all punctuation (; , ' " $ =) must be entered as shown. The non-terminal symbols used are fully defined under "Syntax" at the end of this appendix.

Structure of a resource description file

The 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.

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* '(' *resource-ID* | *resource-name* ')' `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-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* ')' }? '{' { *type-statement* ; }* '}' ;

resource-type ::=	*long-expression*

type-statement ::=	*data-type*
	fill-type
	alignment
	switch-type
	array-type

Data-types

data-type ::=	*data-type-specifier* { *symbolic-declaration* | = *declaration-constant* }?
data-type-specifier ::=	`char`
	`string` { '[' *length* ']' }?
	`pstring` { '[' *length* ']' }?
	`cstring` { '[' *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* }*
range-block ::=	*identifier* { = *declaration-constant* }? { , *identifier* }*
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 ::=	{ data-statement { , data-statement }* }?

Preprocessor Directives

The following preprocessor directives are available.

`#define` *identifier* [*define-string*]? *newline*
`#undef` *identifier newline*
`#if` *preprocessor-expr*
`#elif` *preprocessor-expr*
`#else`
`#endif`
`#ifdef` *identifier*
`#ifndef` *identifier*

preprocessor-expr is the same as *expression* with the following additional expressions:

`defined` '('*identifier*')'
`defined` *identifier*

Syntax

This section defines the non-terminal 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 any length.

Token delimiters

token-delimiter ::=	[*space* | *tab* | *newline* | *comment*]+
comment ::=	'/*' [*printing-character*]* '*/'

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
                         - 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 ')'
```

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

string-variable ::=	`$$Version`
	`$$Date`
	`$$Time`
	`$$Shell'('"`*Shell-variable-name*`"')'`
	`$$Resource'('`*file*, *resource-id*, *resourceName-or-ID*`')'`

	resourceName-or-ID::= *resource-id* *resource-name*

numeric-variable ::=	`$$Hour`
	`$$Minute`
	`$$Second`
	`$$Year`
	`$$Month`
	`$$Day`
	`$$Weekday`

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

Files types and creators

Table E-1 lists MPW file types and creators.

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 '
Pascal load/dump	'DMPP'	'MPS '
Assembler load/dump	'DMPA'	'MPS '

File suffixes

File suffix conventions are as follows.

Text files:

name.a	assembly-language source file
name.a.lst	Assembler listing file
name.p	Pascal source file
name.c	C source file
name.h	C header file
name.r	resource description file (Resource Compiler 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 Pascal Compiler
name.c.o	object file created by the 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.)

Appendix F

Writing an MPW Tool

Shell facilities xx

Parameters x
Shell variables x
Standard I/O channels x
    I/O buffering x
    I/O to windows and selections x
Signal handling x
Exit processing x
Status codes

Restrictions xx

Initialization x
Memory management x
    Heap x
    Stack x

Windows, graphics, and events xx

Conventions xx

This appendix provides information specific to writing an integrated MPW tool. You'll also need to refer to the following:

- "Putting Together an MPW Tool" in Chapter 7 for information about the mechanics of linking and installing a tool.
- The appropriate MPW language reference manual for details of the Integrated Environment routines available in the various language libraries. (The **Integrated Environment** is a set of routines, modeled on the C language, that provide parameter passing, access to variables, and other functions to MPW tools.)

Shell facilities

Tools running within the MPW Shell environment are provided with many facilities, including

- parameter passing
- access to Shell variables
- a set of pre-opened 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

Parameters

Parameters are passed to tools by the Shell. Every program is passed at least one parameter: the name of the program itself. This parameter is always the first parameter (technically, parameter 0) and is useful for error messages or other special action.

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 3.) This text is then split up into individual words and placed in a convenient data structure for programmatic access:

C:	In C, the main program is actually passed two parameters, named `argc`, the argument count, and `argv`, the argument vector. 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 F-1.)
Pascal:	In 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 an array of Pascal string pointers.
Assembly language:	The Integrated Environment routine, _RTInit, can be used to access the command parameters in assembly language. The addresses of `argv` and `argc` are passed to _RTInit, which initializes them. See Appendix I of the *MPW Assembler Reference* for details about this routine.

C and Pascal examples are shown in Figure F-1.

```
C Sample.c -o Sample          Pascal Sample.p -o Sample
```

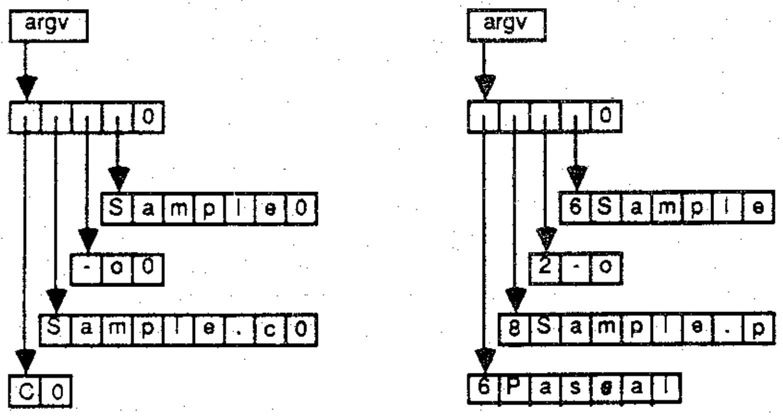


Figure F-1
Parameters in C and Pascal

Environment (Shell) variables

The MPW Shell maintains a set of state variables that can be made available to tools with the Export command. When a tool is run, 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, or
- Using the `getenv` function.

Because only a copy is passed, a tool cannot alter the Shell's value of a variable.

C:	Shell variables are accessible in C via the third parameter to the main program, called `envp` (the environment pointer). `envp` is a pointer to a zero-terminated array of pointers to ***name/value*** C-string pairs. (Each pair is of the form ***name*0*value*0**.) The C library provides the `getenv` routine, which, given a variable name, looks up its value.
Pascal:	Pascal programmers are provided with another IntEnv unit global variable, also called `envp`. The structure used is the same as that for C, except that pointers to strings are forced to even byte boundaries by zero padding, if necessary. To facilitate the lookup of values for given Shell variables, a routine called IEGetEnv is provided in the IntEnv unit.
Assembly language:	The Integrated Environment routine, _RTInit, can 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. You can use `getenv` for C strings, or IEGetEnv for Pascal strings (from the PasLib library). See Appendix I of the ***MPW Assembler Reference*** for details on calling _RTInit.

Standard input/output 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, windows on the screen). Program input may be typed (or selected) and entered; program output appears immediately after the command. This input and output may be taken from or directed to other files by specifying I/O redirection (<, >, >>, ≥, ≥≥) 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 from the console, all 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 diagnostics are being sent to the console, the order of the output on the screen may not match the order in which the data was written, because of differences in when the separate buffers are flushed, as illustrated in Figure F-2. You can circumvent this problem by flushing standard output before writing to diagnostic output.

❖ *Note:* Figure F-2 shows the output conventions in C and Pascal. Assembly-language programmers must do their own buffering, or call C or Pascal routines.

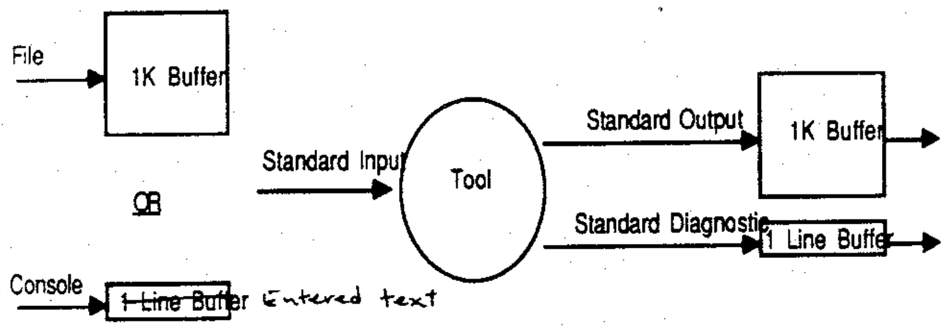


Figure F-2
I/O buffering

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 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. The use of these is documented in detail in the *MPW Pascal Reference*.

I/O to windows and selections

The MPW environment also provides 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.

Signal handling

The MPW environment provides a set of routines to handle signals. A **signal** is an event that diverts program control from its normal execution sequence.

❖ *Note:* The only signal currently supported is Command-period, the standard Macintosh command for terminating the execution of an operation.

Signals can be caught, held and released, and ignored. The default action of any signal is to close all open files, execute any exit procedures (described below in "Exit Processing"), and terminate the program. If, however, your program requires special handling of a signal, or chooses to ignore it, you can use the procedure `sigset` to replace the default signal-handling procedure with your own procedure. Your program can also temporarily suspend action on a signal (for instance, before entering a critical section of code) by calling `sighold`. You can restore the signal by calling the procedure `sigrelease`, whereupon the signal-handling procedure will take affect if the signal was raised during the hold period. Your program may also pause until one or more signals are raised by calling the procedure `sigpause`. See the MPW language reference manuals for the details on how to use these routines.

Exit processing

A program often requires some special processing before terminating. You can use the procedure `onexit` to register a procedure to be called at program termination or when the exit procedure is called. This procedure guarantees your program a chance to do any cleanup before terminating. This is especially useful for cleaning up after an uncaught signal.

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

You may want to return error codes other than these. In that case, you should carefully document the numbers and their meanings.

C:	Result codes are returned from C tools as the function result of the procedure `main()` or by passing them as the parameter to the C Library `exit` function.
Pascal:	Pascal programmers must call the IntEnv procedure IEexit to return the status result.
Assembly language:	The Integrated Environment routine _RTExit is available to assembly-language programmers. _RTExit takes the status code as a parameter.

Restrictions

Tools are similar to desk accessories in that they coexist with another program (the MPW Shell), and many of the same restrictions apply to tools as to desk accessories. (See "Writing Your Own Desk Accessories" in the Desk Manager chapter of *Inside Macintosh.*) The following sections touch on some of the considerations in enabling tools to coexist with the Shell.

Initialization

Caution

Since 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 program 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 no work properly unless InitGraf is called.

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.

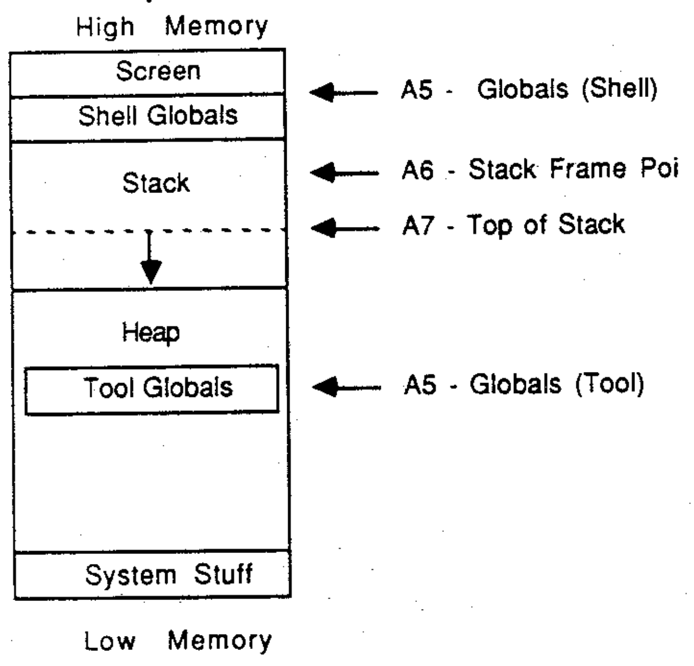


Figure F-3
Memory map

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 been left 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 penalty should the space be needed for other purposes.

Heap

Because the Shell and tools share the same heap, some cooperation is necessary to ensure its efficient use. 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:

- If RAM caching is being used, you can reduce the size of the cache.
- You can free up about 45K by running without the debugger (that is, name it something other than MacsBug and reboot, or hold down the mouse button while booting).
- You can minimize the number of windows open when the tool is run. (Certain memory-resident data structures are required for each window.) Directing program output to a file instead of a window will also provide the tool with more memory.
- You can also reduce the stack space by using the Shell resource described in the next section.

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. Because some tools may require more stack space or more heap space, 'HEXA' resource number 128 is available in the Shell to adjust the maximum stack size.

❖ *Note:* Because the stack is shared between the Shell and the tool, executing tools from within nested command files results in less stack space for the tool. The Shell uses about 100 bytes of stack per nesting level.

Windows, graphics, and events

The creation of windows, use of graphics, and event processing by tools is a largely unexplored area in the MPW environment. MPW aims to support these types of tools; however, little work has been done so far in this area, and unknown restrictions may exist.

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.
- 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 directly executed, to open the file and select the offending line. (See "Executable Error Messages" in Chapter 3.)

See the "Command Prototype" section at the beginning of Chapter 9 for more information on MPW command-language conventions.

Appendix G

Writing a Desk Accessory or Other Driver Resource

This appendix documents the DRVRRuntime library and describes the specifics of writing a desk accessory or other driver with the MPW system. (A desk accessory is a special case of a driver; all of the information in this appendix applies to both.) You should already be familiar with the following:

- "Writing Your Own Desk Accessories" in the Desk Manager chapter of *Inside Macintosh*
- The Device Manager chapter of *Inside Macintosh* (for information about 'DRVR' resources, and so on)
- "Putting Together a Desk Accessory or Driver" in Chapter 7 of this manual

For information about the actual routines used in Pascal, C, or assembly language, see the appropriate MPW language reference manual.

The DRVRRuntime library

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 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 strings, windows, and menus, if used in your desk accessory. (For an example of such a resource description file, see "The Desk Accessory Resource File" in Chapter 7.)

The advantages of using DRVRRuntime.o are the following:

- 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 following 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, the following 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 an INTEGER, and is also defined in OSIntf.p.

In C, you would need to write the routines as follows:

```
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 will reside 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 of your functions cannot be used. In particular, the following language constructs reference the global data area and cannot be used:

Asm:	No DATA directives
Pascal:	No UNIT variables
C:	No `static` or `extern` variables; no `string` constants

Also note that QuickDraw globals cannot be used directly. Further, you cannot call library functions that use any of these things. (Look for the Linker message "No global data was allocated.")

❖ *Note:* Apple is investigating the use of A5-based global variables in desk accessories. Currently several Macintosh applications contain trap-override or ROM hook routines that expect A5 to point to the application's globals, but without saving, setting, and restoring A5 to ensure that this is the case. Such applications are incompatible with desk accessories that use A5, because the desk accessory's calls to the ROM could end up in the application's trap-override or hook code.

Typically, C and Pascal programmers will allocate global storage from the heap and use 'STR#' resources for string constants. If you need to allocate global data from the heap, you can declare a record containing all of the global variables you need. Then, in your DRVROpen routine, you should allocate memory from the heap with the size of this record, and store its pointer (or handle) in the dCtlStorage field of the device control entry. Then, to reference an element in the record, you can use this pointer (or handle) to reference the global variable that 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 bytes free on the default volume, along with the name of the default volume.
- It performs these operations every five seconds, so that you can see how your memory conditions change.

For instructions on building this desk accessory, see the Instructions file in the Examples folder, or refer to "Putting Together a Desk Accessory or Driver" in Chapter 7.

Appendix H

Object File Format

Object file format xx

Notation used in this appendix xx

Object file records xx

Pad record x
First record x
Last record x
Comment record x
Dictionary record x
Module record x
Entry-point record x
Size record x
Contents record x
Reference record x
Computed-reference record x

This appendix is addressed to programmers who are writing compilers or assemblers to run under MPW.

Object file format

An object file consists of a sequence of object file records. These records are in the data fork of the file. There are 11 types of object file records:

- The first record in the file must be a **first 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.
- **Entry-point records** define entry points in code and data modules.
- **Size records** specify the size of a module.
- **Contents records** specify the contents of a module.
- **Reference records** and **computed-reference records** specify locations in modules that contain references to other modules or entry points.
- The last record in the file must be a **last record**.

A **module** is a contiguous region of memory that contains code or static data. (The jump table is considered to be code.) 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.

All modules, entries, and segments are given a unique, positive, 16-bit ID. An **ID** is a file-relative number for a module, an entry point, or a segment, identifying the module, entry point, or segment within a single object file.

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. A module, entry point, or segment without a name is said to be anonymous.

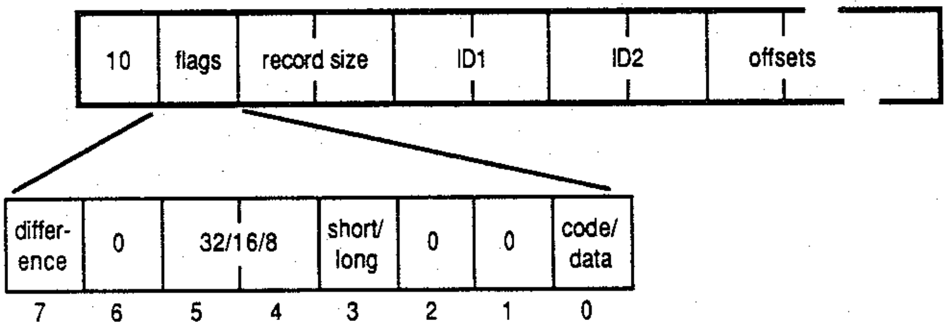
Names and IDs are specified in dictionary records. Local IDs may be anonymous. (If no dictionary entry is found for it, an ID is considered anonymous.) Local modules and entries need not have unique names, and an external segment may have the same name as an external module or entry point.

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 records, a flag bit indicates whether the record refers to the code or the data module.

The structure and semantics of each of the record types is defined below.

Notation used in this appendix

Each record type is represented by a diagram such as the following:



The first box illustrates the record. Each 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 32K 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 a 1 indicates data
1, 2	must always be 0
3	0 indicates short and a 1 indicates long

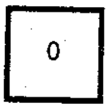
4–5	0 indicates 32 bits, 1 indicates 16 bits, and 2 indicates 8 bits
6	always 0
7	1 indicates a difference computation

❖ All unspecified bits must be zero.

Object file records

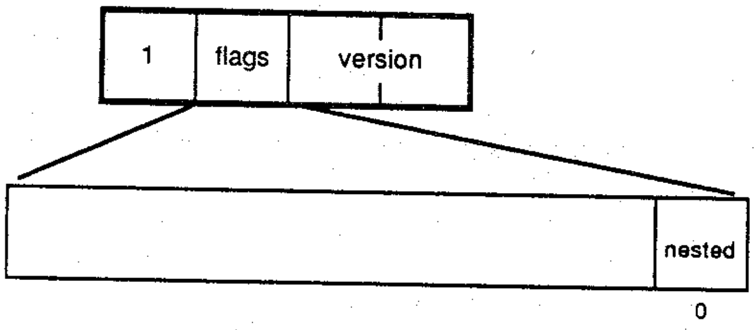
This section defines each of the object file record types.

Pad Record



A pad record is a single byte that is always zero. A pad record follows any record whose length is an odd number of bytes, in order to maintain word alignment. (Other than pad records, all records are word-aligned.)

First record

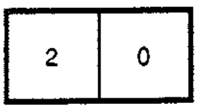


The first record in an object file must be a first record.

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.

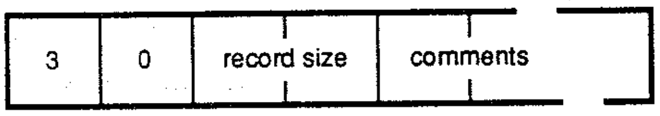
The *version* field contains a version number that is 1 for the current definition of the object file format.

Last record



The last record in an object file must be a last record.

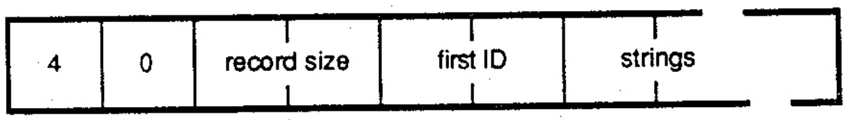
Comment record



A comment record allows comments to be included in an object file. It has no effect on the semantics of the object file.

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.

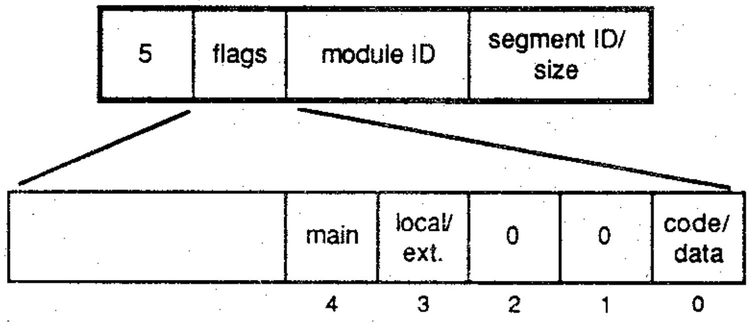
The *record size* field specifies the total number of bytes in the record.

The *strings* field contains one or more names, each of which is preceded by a length byte.

The first name in the strings field is associated with the ID given in the *first ID* field. The second name is associated with *first ID*+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.

Module record



A module record associates an ID with a module, and makes that module the current code or data module. All entry-point, size, contents, reference and computed-reference records help define the current code or data module.

Modules may contain either *code* or *data*:

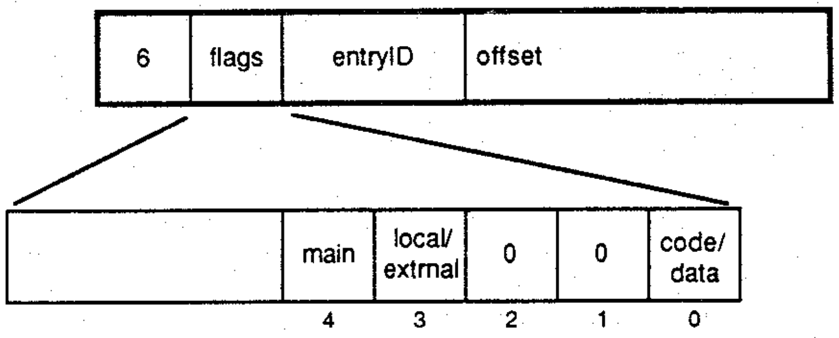
- For code modules, the *segment ID* field specifies the segment in which the code is placed. Segments may be named or anonymous. Named segments are treated as external; anonymous segments are local. (If the segment is named, the dictionary record specifying the name must appear before the segment ID 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 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.

References to a module are considered to be references to the first byte of the module.

Entry-point record

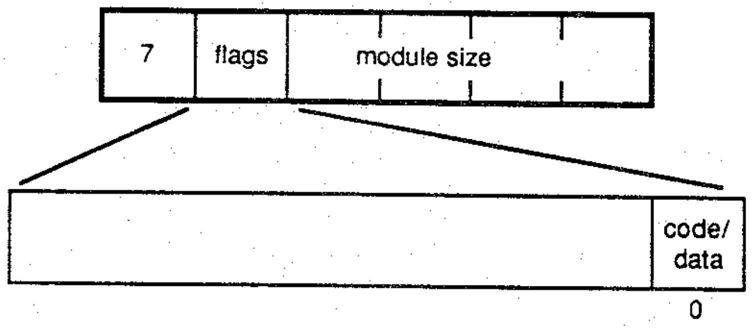


An entry-point record declares an entry-point ID. The entry point is in the current code or data module, as indicated by bit 0 of the flags field.

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).

Flags: An entry points 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.

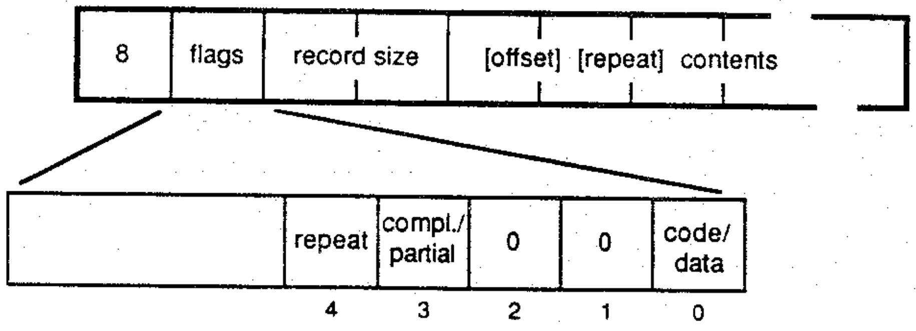
Size record



A size record specifies the size of the current code or data module. The size is in bytes. The bytes within a module of size *N* are numbered 0, 1, ..., *N*-1. 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.

❖ *Note:* In allocating records, the Linker rounds the size of a module up to a multiple of two, to ensure that modules are word-aligned in memory.

Contents record



Contents records specify the contents of the current code or data module.

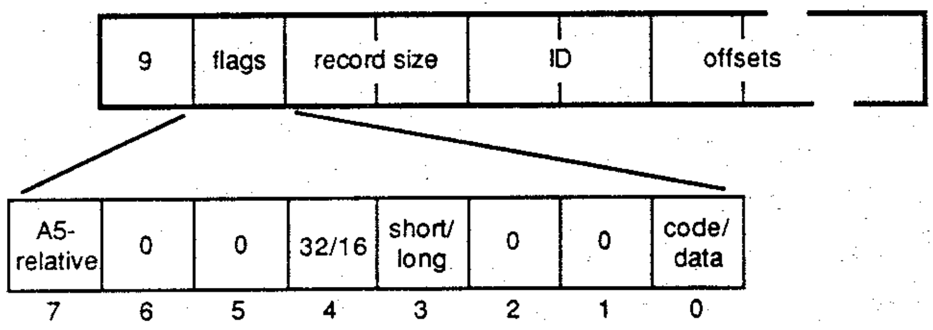
The *record size* field specifies the total number of bytes in the record.

Either *complete* or ***partial*** contents may be specified. If partial contents are specified, the first four bytes of the contents field specify the *byte 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.

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 four categories: from code to code, from code to data, from data to code, and from data to data.

- **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 is assumed to 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.

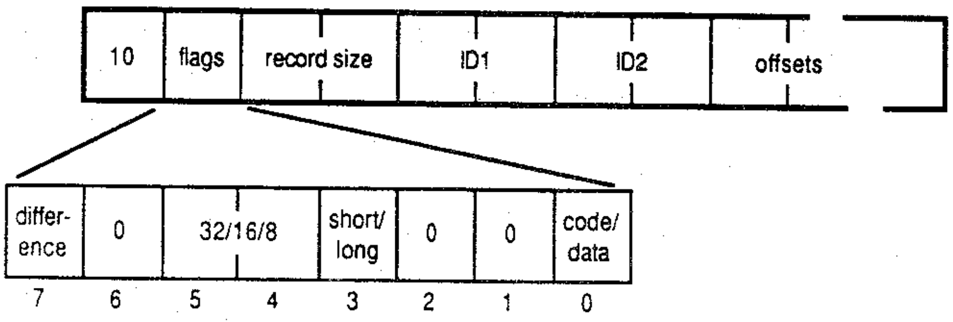
In either case, the location may be 32 or 16 bits. (32-bit PC-relative and A5-relative address modes are available for the 68020, but not for the 68000.)

- **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 32 or 16 bits. (32-bit A5-relative addressing is available for the 68020, but not for the 68000.)
- **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 32 or 16 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.)

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***.

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).

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.

Appendix I

In Case of Emergency

This appendix contains some information that may be useful when serious system errors occur.

Crashes

If you end up in the debugger (MacsBug) while running MPW, it may be possible to recover without rebooting and losing your recent changes. The debugger displays the register contents followed by a ">" prompt. If a tool is being executed, type `G STOPTOOL` and press Return to return to the Shell. If the Shell is being executed, 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 either of these steps 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 the only 'HEXA' resource in the file MPW Shell. The default size is $2710 (10,000 bytes). Doubling this to $4E20 (20,000 bytes) has been sufficient for the largest cases we've seen.

❖ *Note:* Increasing the stack size on a Macintosh 512K may create other problems because of decreased heap space.

Glossary

active window: The frontmost window. The Shell variable {Active} always contains the name of the current active 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.

blank: A space or a tab character (in the context of separating words in the command language).

build commands: Shell commands that are output by the Make tool, used to build a program.

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.)

code resource: A **resource** that contains a program's code—most commonly a resource of type 'CODE' (for applications and MPW tools), but other resource types such as 'DRVR' and 'PDEF' also contain code.

command file: An ordinary text file (type TEXT) containing a series of commands. The entire file can be executed by entering the filename. Also called a 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 substitution: The replacement of a command by its output. Command substitution takes place within back quotes (`...`).

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.

dependency file: A **makefile**.

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 output to diagnostic output (by default, the active window). You can redirect diagnostic output to another file, window, or selection with the ≥ and ≥≥ operators.

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 reference: A reference to a routine or variable defined in a separate compilation or assembly.

filename: A sequence of up to 31 printing characters (excluding colons), which identifies a file. More at **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.

HFS: "Hierarchical File System": used on 800K disks and the Apple Hard Disk 20.

insertion point: An empty selection range; that is, the character position where text will be inserted (marked with a blinking vertical bar).

interface routine: A routine called from Pascal 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.

main segment: The **segment** containing the main program.

makefile: A file used by the Make command, which 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.

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).

non-HFS: The "flat" 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 dash (-).

pathname: A sequence of up to 255 characters that identifies a file or directory. A *full pathname* is a pathname that contains embedded colons but no leading colon. A *partial pathname* either contains no colons or has a leading colon.

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 3, Table 3-1.)

position: In editing commands, position refers to the location of the **insertion point**.

prefix: The directory portion of a filename.

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.

regular expressions: A language for specifying text patterns, using a special set of metacharacters. (See Appendix B, Table B-2.)

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.

script: A **command file**.

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.)

standard error: Diagnostic output.

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 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.

target selection: The current selection in the target window, represented by the § character.

target window: The second window from the top—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.

tool: An **MPW tool**.

word: A single, blank-separated element in a command. A command name and each of its parameters are separate words in the command language.

MPW & MacApp Bug Report Form

BACKGROUND

Date ____________________ Version ____________________

AREA: Compiler: C Pascal

Assembler

Library: C Pascal Assembly

MacApp

Shell/Editor

Tool ____

Performance

BUG DESCRIPTION

____

____

____

____

____

____

____

____

____

CONTACT INFORMATION

Name: ____________________ Phone/Ext. ____________

Address: ____

City, State, Zip ____________________________________

Please return completed form to:
MPW Bug Report; MS 27S; Apple Computer, Inc.; 20525 Mariani Ave.; Cupertino, CA 95014