New Technology Building Blocks

DISK INCLUDED

Macintosh Programming Techniques

A Foundation for All Macintosh Programmers

- Learn essential techniques for expert Macintosh programming
- Understand terms and concepts unique to Macintosh programming
- Gain hands-on experience with interactive software tutorial

Dan Parks Sydow

M&T BOOKS

Series Editor: Anthony Meadow
Bear River Associates

Macintosh Programming Techniques

A Foundation for All Macintosh Programmers

Macintosh Programming Techniques

A Foundation for All Macintosh Programmers

Dan Parks Sydow

M&T Books
A Division of MIS:Press, Inc.
A Subsidiary of Henry Holt and Company, Inc.
115 West 18th Street
New York, New York 10011

Printed in the United States of America

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging-in-Publication Data

Sydow, Dan P.
Macintosh programming techniques: a foundation for all Macintosh programmers / Dan P. Sydow
p. cm.
Includes index.
ISBN 1-55828-326-9 : $34.95
1. Macintosh (Computer)--Programming. I. Title.
QA76.8.M3S96 1993
005.265--dc20 93-46619
CIP

96 95 94 4 3 2

Publisher: Steve Berkowitz
Associate Publisher: Brenda McLaughlin
Project Editor: Margot Owens Pagan
Development Editor: Mike Miley
Copy Editor: Judy Whittle
Technical Editor: Ray Valdés
Production Editor: Eileen Mullin
Associate Production Editors: Maya Riddick, Joseph McPartland
Cover Design: JPD Communications

Dedication

To my wife, Nadine...

Dan

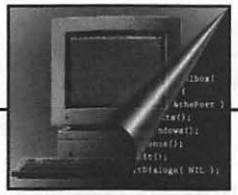


Table of Contents

Foreword xxi

Why This Book is for You 1

Introduction 3

What's on the Disk 4
What You Need 5
Extracting the Contents of the Disk 5
Using the In Action! Program 9
    Running the program 9
    The program windows 9
    Moving about in the program 10

Chapter 1: Introduction to Macintosh Programming 13

Development Systems 13
    Information environments 14
    Application frameworks 14
    Programming languages 14

About Macintosh Programming 15
Bit-mapped Graphics 16
Event-driven Programming 17
Resources 22
The Toolbox 26
The Operating System 31
System Software 32
The System File and Finder 33
    The System file 33
    The Finder 34
Chapter Program: Intro to Mac Programming 35
    Program project: *VeryBasics.π* 36
    Program resources: *VeryBasics.π.rsrc* 37
    Program listing: *VeryBasics.c* 38
    Stepping through the code 40
        Where are the *#includes*? 40
        Function prototypes 41
        The *#define* directives 41
        Global variables 42
        The *main()* function 42
        Toolbox initialization 42
        Loading a window 43
        Drawing to a window 44
        The main event loop 45
        A Macintosh function 46
Chapter Summary 47

Chapter 1 Lessons on Disk
    Lesson 1–1: Events 22
    Lesson 1–2: Resources 26
    Lesson 1–3: The Toolbox 30

Chapter 2: Macintosh Memory 49

Memory Organization 49
    System partition organization 50
        System global variables 51
        System heap 51
    Application partition organization 52
        A5 World 52
        Application stack 52

Application heap 54
Summary of memory organization 56
The Application Heap 57
Heap fragmentation 58
Heap compaction 60
Nonrelocatable and relocatable blocks 61
Chapter Program: Memory Partitions 65
Program resources: *MemoryFiller.π.rsrc* 68
Program listing: *MemoryFiller.c* 69
Stepping through the code 70
Chapter Summary 71
Disk Files
Lesson 2–1: Memory organization 57
Lesson 2–2: Heap fragmentation 58
Lesson 2–3: Heap compaction 60
Lesson 2-4: Master Pointers and Handles 64

Chapter 2 Lessons on Disk
Lesson 2–1: Memory organization 57
Lesson 2–2: Heap fragmentation 58
Lesson 2–3: Heap compaction 60
Lesson 2–4: Master pointers and handles 64

Chapter 3: Resources 73

About Resources 74
The importance of resources 74
Resource types 75
Checking for errors 75
Working With Strings 77
The 'STR#' resource 77
Using a string in a program 78
Pictures and Animation 81
The 'PICT' resource 81
Displaying a 'PICT' in a program 83
Using 'PICT's to create animation 86
Creating a series of 'PICT's 86
Animation source code 88
Sounding Off 91
The 'snd ' file 91
The 'snd ' resource 92

Playing a 'snd ' in a program 94
Giving a Program an Icon 95
The Finder and icons 96
Creating the 'BNDL' resource 97
Setting the Creator in the compiler 102
Making the Finder aware of a new icon 102
Chapter Program: Using Resources 103
Program resources: *ResourceUser.π.rsrc* 104
Program listing: *ResourceUser.c* 106
Stepping through the code 110
The *#include* directives 110
The *#define* directives 110
The *main()* Function 111
Using strings 112
Using pictures for animation 113
Playing sounds 113
Chapter Summary 113

Chapter 3 Lessons on Disk
Lesson 3–1: The 'STR#' resource 80
Lesson 3–2: The 'PICT' resource 86
Lesson 3–3: Animation 90

Chapter 4: QuickDraw Graphics 115

About QuickDraw 115
Initializing QuickDraw 116
Pixels and the coordinate system 117
Graphics Ports 118
The *GrafPort* and *GrafPtr* 118
The graphics pen 119
Defensive Drawing 121
Changing ports 121
Changing characteristics of a port 123
Drawing Shapes 125
Working with rectangles 126
Working with ovals 128
Working with round rectangles 129
Patterns 131
The 'PAT ' resource 131
The pattern source code 133

Color QuickDraw 134
Checking for color 134
Color windows 136
Color patterns 137
The 'ppat' resource 138
The color pattern source code 138
Color drawing 140
The 'wctb' resource 141
The Cursor 143
Chapter Program: Drawing on the Mac 146
Program resources: *QuickDrawing.π.rsrc* 146
Program listing: *ResourceUser.c* 147
Stepping through the code 152
The *#include* directives 152
The *#define* directives 152
The *main()* function 153
Initialization 153
Checking for color 154
Preserving the environment 154
Lines and shape painting 155
Shape filling 156
Chapter Summary 157

Chapter 4 Lessons on Disk
Lesson 4–1: Moving the pen 121
Lesson 4–2: Switching ports 123
Lesson 4–3: Using patterns 134

Chapter 5: Working With Windows 159

Windows Primer 160
The WIND resource 160
Loading a 'WIND' 160
The WindowRecord, WindowPtr, and WindowPeek 161
Event Handling 163
Windows and Events 170
Mouse down events 171
Handling a mouse click in a drag bar 171
Handling a mouse click in a close box 173
Handling a mouse click in a content region 174
Handling a mouse click in other areas 175

Single Window Techniques 176
Activate events 177
Updating a window 177
Simple window techniques 182
Moving a window 183
Showing and hiding a window 184
Changing a window's title 184
Multiple Window Techniques 184
Expanding the WindowRecord 185
Activates and multiple windows 190
Updates and multiple windows 192
Chapter Program: Working With Multiple Windows 193
Program resources: *MultiWindows.π.rsrc* 194
Program listing: *MultiWindows.c* 195
Stepping through the code 205
The *#include* directives 205
The *#define* directives 205
Global types 206
Global variables 206
The *main()* function 207
Initialization 208
Marking and examining a window 209
Opening a window 211
Event handling 213
Chapter Summary 220

Chapter 5 Lessons on Disk
Lesson 5–1: Handling events 169
Lesson 5–2: Window updating 182
Lesson 5–3: *MyWindPeek* 185
Lesson 5–4: Using *MyWindPeek* 185

Chapter 6: Dealing With Dialogs 223

Alerts 224
Alert resources: 'ALRT' and 'DITL' 224
Alert source code 227
Dialogs 228
Dialog Resources 229
Dialog item types 229
The 'DLOG' and 'DITL' Resources 230

Dialog Items 236
Getting dialog item information 237
Working with edit text items 238
Working with check box items 239
Working with radio button items 240
Modal Dialogs 241
The *DialogRecord* 242
Modal dialog source code 244
Modeless Dialogs 246
Using User Items 250
The user item resource 251
The user item source code 251
Color Dialogs 258
Chapter Program: *DialogPlus* 259
Program resources: *DialogPlus.π.rsrc* 260
Program Listing: *DialogPlus.c* 263
Stepping through the code 273
The *#define* directives 273
The global variables 274
The *main()* function 274
Handling check boxes and radio buttons 275
Opening a window and a modeless dialog 276
Drawing user items 278
Event handling 279
Chapter Summary 285

Chapter 6 Lessons on Disk
Lesson 6–1: The *DialogPtr* 243
Lesson 6–2: Using modal dialogs 246
Lesson 6–3: Using modeless dialogs 250

Chapter 7: Managing Menus 287

About Menus 287
Menu Resources 289
The 'MENU' resource 289
The 'MBAR' resource 291
Menu Source Code 292
Setting up the menu bar 293
Handling a click in a menu 296
Handling a click in the Apple menu 300

Handling a click in other menus 301
Keyboard Equivalents 302
The 'MENU' resource 303
Handling a keystroke 304
Hierarchical Menus 305
The 'MENU' resource 306
Setting up the hierarchical menu 308
Changing Menu Characteristics 311
Disabling and enabling menus and menu items 311
Adding a checkmark to a menu item 315
Changing the text of a menu item 318
Changing the style of a menu item 320
Editing Text in a Modal Dialog 323
Checking for System 7 324
Modal dialog filter function 325
Chapter Program: MenuMaster 331
Program resources: *MenuMaster.π.rsrc* 335
Program listing: *MenuMaster.c* 339
Stepping through the code 349
The *#define* Directives 350
The global variables 352
The *main()* function 352
Initializing variables 353
Setting up the menu bar 353
Handling a keystroke 354
Handling a click in the menu bar 356
Editing text in a modal dialog 359
Checking a menu item 361
Disabling and enabling a menu and menu item 361
Handling a hierarchical menu 363
Chapter Summary 364

Chapter 7 Lessons on Disk
Lesson 7–1: The Menu bar 296
Lesson 7–2: Filter functions 331

Chapter 8: The Varying Mac 367

Checking for Traps 368
Toolbox routines are traps 368
Determining if a Toolbox routine is implemented 375

The Features of a Macintosh 378
More Features of a Mac 379
The *Gestalt()* Function 379
Checking for the availability of *Gestalt()* 380
Determining machine features using *Gestalt()* 382
Determining the QuickDraw version 385
Determining the CPU type 386
Determining the amount of physical RAM 386
Determining the floating-point coprocessor type 387
Determining the Macintosh machine type 388
Determining the operating system version 389
Monitor-Aware 390
Dealing with multiple monitors 390
Setting the window drag region 390
Setting the center point for windows 392
Dealing with different sized monitors 396
Color-Aware 396
Color representation 396
Getting the pixel depth of a monitor 398
Multiple-monitors and pixel depth 400
When to call the pixel depth routines 404
Chapter Program: *InnerView* 406
Program resources: *InnerView.π.rsrc* 407
Program listing: *InnerView.c* 408
Stepping through the code 420
The *#include* directives 420
The *#define* directives 420
The global variables 422
The start 422
Checking the system 423
Putting up the menu 426
Opening a window 426
Event handling 427
Closing a window 435
Chapter Summary 436

Chapter 8 Lessons on Disk
Lesson 8–1: Traps and the Toolbox 374
Lesson 8–2: Pixel depth 400

Chapter 9: Memory Management 437

Macintosh Memory Management 438
Avoiding Heap Fragmentation 439
How nonrelocatable blocks are created 440
Reserving memory to reduce fragmentation 441
Reserving Memory 446
Allocating master pointer blocks 446
Setting the heap size 449
Writing 32-bit Clean Programs 450
Segmentation 452
Segmenting a program in THINK C 455
Determining how to segment a program 459
The main segment 461
Unloading segments 467
Setting a Program's Size 471
The user's role in setting the partition size 471
Setting an application's partition size in THINK C 473
Using the 'SIZE' resource to set a partition 474
Determining your application's memory needs 476
Watching program memory using the Finder 477
Watching program memory using *Swatch* 480
Handling Memory Errors 481
Watching for failed memory allocations 482
Providing the user with error information 483
Chapter Program: Tying it All Together 485
Program resources: *InnerViewII.π.rsrc* 488
Program THINK C project: *InnerView.π* 494
Program listing: *InnerViewII.c* 498
Defines.h 498
Globals.h 500
Initialize.h 501
Utilities.h 502
Initialize.c 502
Utilities.c 507
InnerViewII.c 509
Stepping through the header files 525
Defines.h 525
Globals.h 526
Initialize.h and *Utilities.h* 527
Stepping through *Initialize.c* 527

Checking the system 527
Reserving memory 528
Initializing variables 528
Opening the window and dialog box 529
Stepping through *Utilities.c* 531
Stepping through InnerViewII.c 531
The *main()* function 531
Dimming a menu 534
Updating the window 535
Getting machine information, and error handling 538
Bits, masks, and Str255 539
Displaying machine information 542
Displaying a picture 545
Chapter Summary 546

Chapter 9 Lessons on Disk
Lesson 9–1: Reserving memory 445
Lesson 9–2: Segmentation 471

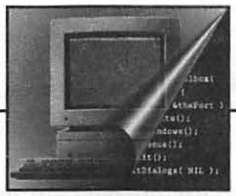
Appendix A: Macintosh C Data Types 549

Appendix B: Determining a Trap's Type 553

Appendix C: Gestalt Definitions 557

Appendix D: Toolbox Routine Summary 567

Index 605



Acknowledgments

Anthony Meadow, Bear River Group, for kicking things off.

Carole McClendon, WaterSide Productions, for introducing me to M&T Books.

Mike Miley, for a number of helpful comments during editing, and for playing middleman to keep things rolling.

Ray Valdés, for numerous suggestions made during editing. A special thanks to Ray for tactfully saying, "this is not quite accurate" when I was dead wrong!

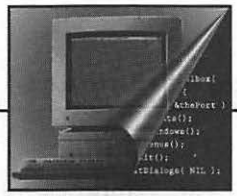
Judy Whittle, TechnoMark, for making me look like I write in a grammatically correct style.

Eileen Mullin, M&T Books, for the effort put into laying out 600 pages, and for giving the book such a polished appearance.

Maya Riddick and Joseph McPartland, M&T Books, for cleaning up many of the figures that appear in the book—I know a lot of work went into this.

Margot Pagan, M&T Books, for putting up with my unorthodox way of doing things.

Joe Holt, Adobe Systems, for granting permission to include his simple and elegant Macintosh program, Swatch, with this package.



Foreword

The Macintosh operating system (which, surprisingly, does not have a name) is complex: there are more than 2000 system calls available in System 7.0, and the number is growing. In fact, the Macintosh OS is as complex as any newer operating system, such as UNIX, and is far more complex than earlier personal computer operating systems, such as MS-DOS. One reason for this complexity is Apple's strategy to add new sets of features continuously, providing new services for developers. Since System 7 was released, Apple has shipped additional system software extensions, including QuickTime, Apple Open Collaboration Environment, and AppleScript. Each of these extensions features hundreds of API (Application Programming Interface) calls for programmers to use.

The result? The Macintosh OS is now so complex that no one person can understand it all. This is especially true for programmers who are new to the Macintosh. They need an overview of the whole operating system without getting lost in all the details.

This book provides just such an overview of the Macintosh OS for working programmers. It covers all the important concepts of the basic operating system without trying to explain everything. It is appropriate for programmers with diverse backgrounds, Macintosh users who are learning

how to program the Macintosh, Windows programmers who have to do cross-platform development, or programmers used to other operating systems such as UNIX. It's also useful for people who are not programmers. If you manage programmers, test software, and need to understand how Macintosh applications work, this is about the only book on the market today that gives a technical explanation of how Macintosh applications work with a clarity even non-programmers can understand.

Sophisticated programmers can learn from this book too. Today, more and more programmers are using object-oriented languages to do Macintosh (and Windows) development. Object-oriented programming (OOP) does provide some excellent ways of managing the complexity of operating systems such as the Macintosh's. Nonetheless, anyone programming on the Macintosh still needs to know the basics of the Macintosh OS because even an object-oriented programmer will still have to make many calls to the underlying operating system.

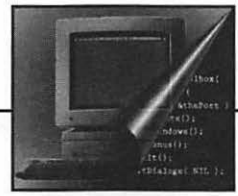
Furthermore, the Macintosh OS works differently than almost every other operating system. For example, memory is managed primarily by using handles, which are pointers to pointers, rather than using just pointers. It is extremely important for any programmer to have a thorough understanding of the implications of this fact, because more bugs in Macintosh applications arise in this area than anywhere else.

What's more, the concept of resources, now commonplace, originated with the Macintosh. Resources have enabled the Macintosh to be highly configurable by programmers and users. It has also allowed the Macintosh to be easily internationalized. The Macintosh operating system is now available in about 40 languages, including English, German, French, Greek, Russian, Chinese, Hebrew, Japanese, Korean and Arabic. A good understanding of resources is critical to good Mac programming

Finally, Macintosh applications are event-driven. This means that the user feels like he or she is in control of the software, and not forced to use a rigid system of menus or commands. This makes it harder for programmers. Users can get to one place in an application from many others, and Macintosh applications have few, if any, modes where user actions are interpreted *only* according to the current mode. In other words, modal applications are easier to write, but harder to use. Macintosh programs are harder to write, but easier to use.

This book, and the software which accompanies it, will help you learn the fundamental concepts of Macintosh programming. The book is a model of clarity and order. When you're finished, you'll have a solid foundation for programming the Macintosh.

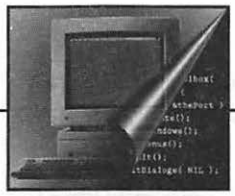
—Anthony Meadow,
Bear River Group



Why This Book is for You

If you've programmed in C, and perhaps have even written programs for Windows, or tried your hand at writing a Macintosh program (with less than desirable results), this book starts you out with the basics of Macintosh programming. *Macintosh Programming Techniques* doesn't make a lot of assumptions about what you already know, it teaches you practical techniques that you will use again and again in all the Macintosh programs you'll be writing.

This book is for anyone interested in learning the techniques and concepts basic to writing programs that will run on the Macintosh. The example code in the book is in the C language. This book does not teach C; it is assumed that you already know either C or C++. The book doesn't, however, assume you've used either of these languages on the Macintosh.



Introduction

Chapter 1 is an introduction to the basic concepts you need to know in order to program on the Macintosh. If you haven't programmed on the Mac, you'll appreciate the definitions of Mac terminology. If you have programmed the Mac, this chapter serves as a refresher. Chapter 1, like every chapter in the book, ends with an example program.

Chapter 2 introduces you to the elementary organization of memory in the Macintosh. Concepts and terms covered here will pop up throughout the remainder of the book.

Chapter 3 discusses resources. Here you'll look at how text, pictures, and even sounds can be stored easily in a resource file. You'll then see how to make use of each of these resource types within your programs. You'll also see how to give your program its own unique icon.

Chapter 4 covers QuickDraw—the Macintosh way of drawing. You'll see how to draw shapes and patterns, in both monochrome and color. This chapter also demonstrates how to add color to the content and title bar of a window.

Chapter 5 discusses windows. Here, basic window management techniques, such as dragging and closing a window, are covered. A large part of this chapter is devoted to the handling of multiple windows.

Chapter 6 covers dialog boxes and alerts. This chapter describes the items that appear in a dialog, including the powerful but seldom-discussed user item. Here you'll see how to work with both stationary (modal) dialogs and movable (modeless) dialogs.

Chapter 7 shows you how to manage menus. You'll see how to define menus using resources and then how to change the characteristics of menus within your source code. After reading this chapter you'll be able to enable and disable menus, change the text of menu items, and add checkmarks to menu items.

Chapter 8 covers the important topic of writing programs that are compatible with the many Macintosh models and configurations now on the market. You'll see how to write programs that will run properly on both monochrome and color Macs, and on Macs with System 7 and pre-System 7 software.

Chapter 9 delves deeper into memory. Here you'll learn about how objects are placed in memory, and what control you have in this process. Many program errors are caused by memory problems. In this chapter I cover error-handling techniques that will make you popular with program users. You'll also learn how much memory you should devote to your final program.

Each chapter ends with an example program. You start out with a simple program that uses just the basics. By the end of the book you'll be comfortable with the example program of Chapter 9—a complete application that works with memory, menus, dialogs, and windows and has its own icon.

What's on the Disk

The disk that is bundled with this book has three folders on it. One folder contains the source code files, resource files, and THINK C project files for all nine of the programs presented in this book. It also contains the standalone version of all nine programs, each ready for you to run.

The second folder contains a program called *In Action! Mac Techniques*. This Macintosh program, written specifically to accompany this book, reinforces the techniques you'll read about. It displays over thirty animated scenes that bring to life the concepts in this book.

The third folder contains a Macintosh utility called Swatch, which allows you to "look inside" your own Macintosh. This small program watches your Mac as it runs. It displays and constantly updates interesting and important information about the memory used by each program.

What You Need

To understand this book you should be familiar with a higher-level language—preferably C or C++.

All you need to run the nine example programs included on the disks is a Macintosh computer that has a 1.4 Mb floppy drive. Apple calls this a SuperDrive, and all of the newer Macinotsh models have it. If you want to edit, modify, and recompile the included source code, you'll need either the THINK C or Symantec C++ compiler. The project files for the nine programs are THINK C 6.0 compatible.

The *In Action! Mac Techniques* program is also ready to run. It runs on any Macintosh that has System 6.0.4 or later, including System 7. It runs on a monochrome or color Mac. Your Macintosh needs 1 M of memory or more to run it.

Extracting the Contents of the Disk

The disk included with this book is a Macintosh-formatted 1.4 Mb disk that contains a single compressed file. This file is self-extracting; you do not have to own any special program to extract the several files and folders that are compressed into this one file.

Copy the file from the disk to your hard drive, then decompress it. You do not have to create any folders on your hard drive; just copy the file directly to your hard drive. Insert the disk into your floppy drive and copy the one file that is on this disk, *MacProgTech.sea*, to your hard drive.

Before decompressing, store the original floppy disk in a safe place. You won't need it anymore, but you'll want to save it as a backup disk.

To decompress, or extract, the files, double-click on the *MacProgTech.sea* icon that is on your hard drive. You'll see the dialog box shown in Figure 1. The names in the list will differ from those in Figure 1 because they will match the names of files and folders on your disk drive.

Click the Extract button to start the extraction process. You do not have to select any file or folder in the dialog list.

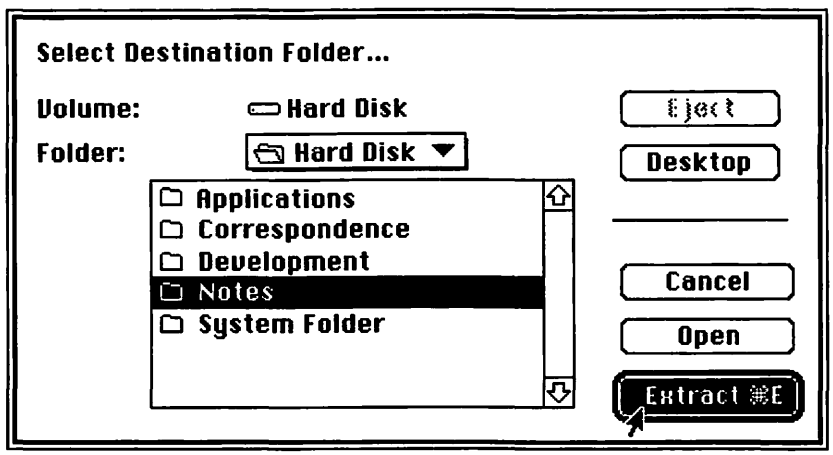


Figure 1. *Dialog you'll see after double-clicking on* MacProgTech.sea

Extraction of all of the compressed files will start. The progress will be displayed in a dialog like that shown in Figure 2.

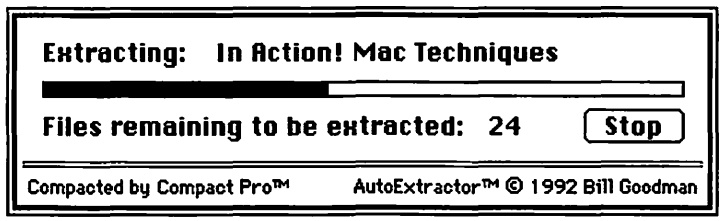


Figure 2. *Extracting the compressed files*

When extraction is complete, you'll have a new folder on your hard drive titled Mac Techniques Folder. That folder will contain three other folders, shown in Figure 3.

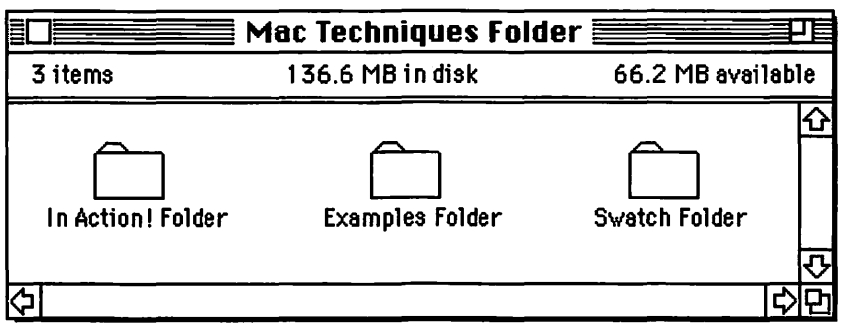


Figure 3. *The contents of the folder after extraction*

You can now drag the *MacProgTech.sea* icon to the Trash can. You've extracted all the files from it, so it is no longer needed. If you accidentally delete files or folders, you can start over again by copying the compressed file from the original disks on to your hard drive.

The folder titled In Action! Folder contains the program *In Action! Mac Techniques*, written specifically to accompany this book. It contains over 30 animated sequences that "bring to life" many of the concepts of this book. This program is described in more detail later. Figure 4 shows what is in the In Action! Folder.

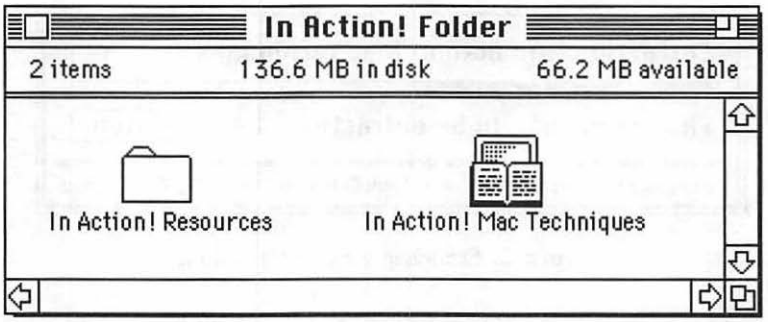


Figure 4. *The contents of the In Action! folder*

You can place the In Action! Folder anywhere you want on your hard drive, but keep both the *In Action! Mac Techniques* program and the folder named In Action! Resources in it. The program uses the contents of this folder and expects it to be right nearby!

The second of the three folders in the Mac Techniques Folder is the Examples Folder. In here you'll find a separate folder for nine Macintosh programs, corresponding to the examples of each of the book's nine chapters. In each of the nine folders you'll find the source code, the resource file, the THINK C project file, and the compiled, ready-to-run application for the example program. If you don't own THINK C or Symantec C++, you can still run each of the programs. If you do own one of these compilers, I've saved you plenty of typing! You can open any of the projects and view or modify the code and recompile it yourself. Figure 5 shows the nine folders found in the Examples Folder.

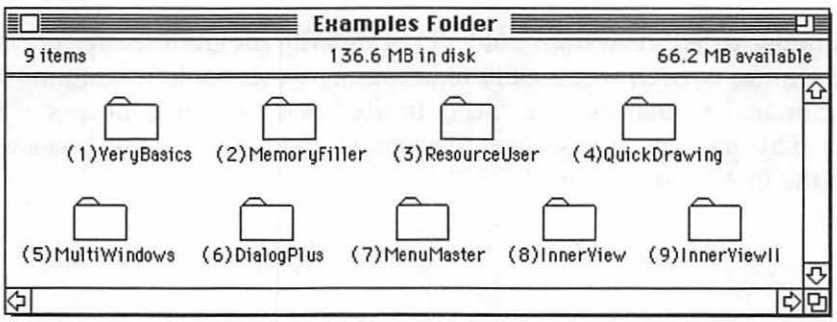


Figure 5. *The contents of the Examples Folder*

The last of the three folders in the Mac Techniques Folder is the Swatch Folder. This folder contains a small Macintosh utility that allows you to view what's going on in the RAM of your Macintosh as programs run. This utility is described in Chapter 9 and in the short text file that is in the Swatch Folder.

Using the In Action! Program

This book endeavors to do a thorough job of explaining the important concepts of Macintosh programming. But the written page sometimes can't do justice to the explanation of certain ideas. The *In Action! Mac Techniques* program was written to fill these voids and specifically to accompany this book.

Running the Program

To run *In Action! Mac Techniques*, simply double-click on its icon as you would any Macintosh program.

After viewing and dismissing the introductory dialog, you'll see two windows on your screen. They're described next.

The Program windows

The In Action! program always has two windows on the screen. The larger of the two contains text, the smaller contains five icons. The smaller window controls the larger window; by clicking on icons in the smaller window you can cause the larger window to display different text. Think of the small window as a control panel that allows you to turn pages in an on-screen book. Figure 6 shows what a typical screen looks like.

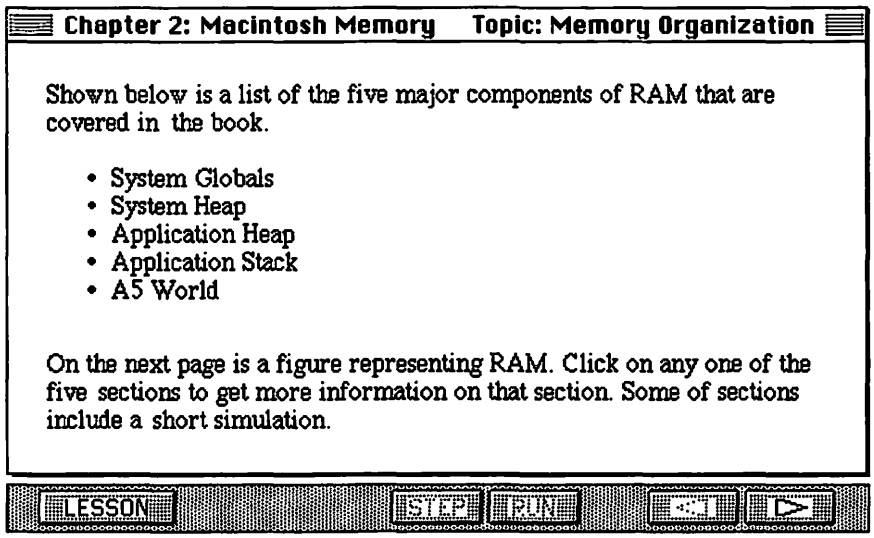


Figure 6. *A view of the two windows used by In Action!*

Moving about in the program

To move to a different screen—to change the contents of the large window—click on either of the two icons on the far right of the control panel. See Figure 7.

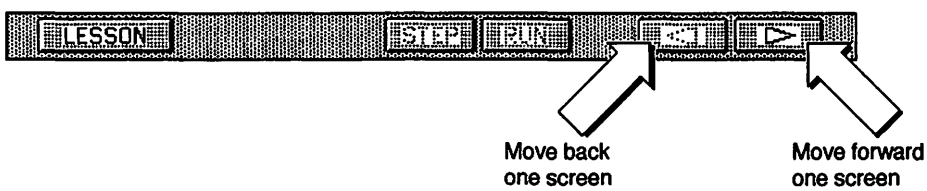


Figure 7. *The control panel icons that allow you to change windows*

The purpose of In Action! is to provide enhanced understanding of the topics in the book. If the program did nothing more than display text, it wouldn't be much of a supplement to the book. To make it worth including, In Action! has to be able to do something a book can't do. That something is to display animation. In Action! contains over 30 short animated sequences that bring to life the figures you'll find in the book.

Many of the program's windows contain animated sequences. You'll recognize a window that has such a sequence by the STEP and RUN icons in the control panel. Normally, they are both dimmed. When you reach a window that contains animation, however, they both become active, or enabled. Click on the RUN icon to view the animation. Or, you can click on the STEP icon to see just a single step of the animation. Clicking on the STEP icon repeatedly will step you through the entire animation. Figure 8 shows a typical window that contains an animation.

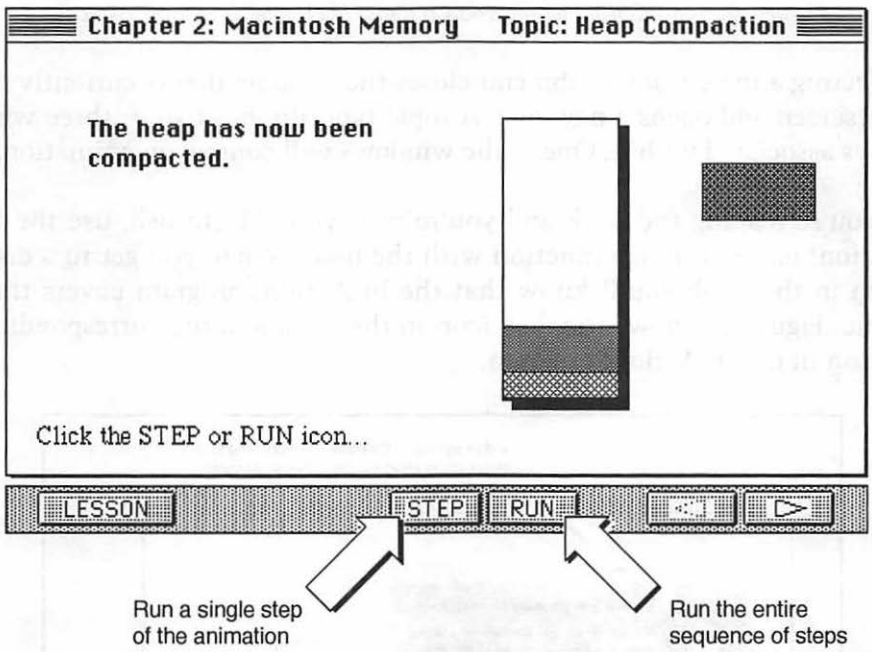


Figure 8. *The STEP and RUN icons allow you to view an animation*

The windows of In Action! are grouped to match the chapters in this book. The last of the five icons on the control panel is used for moving from one chapter to another. When you click on the LESSON icon a menu will drop down. It displays the names of each of the nine chapters found in the book. If you move the mouse over a chapter name, then move to the right, a submenu will be displayed. This submenu lists the topics that In Action! covers from the chapter in the book. Figure 9 shows that In Action! covers four topics from Chapter 2, Macintosh Memory.

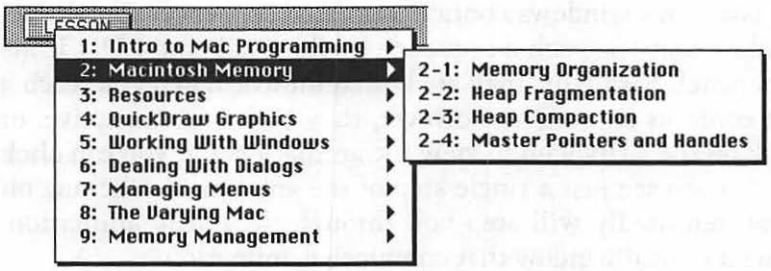


Figure 9. *The LESSON icon reveals a menu of chapter and topic choices*

Selecting a topic from a submenu closes the window that is currently on the screen and opens a new one. A topic typically has two or three windows associated with it. One of the windows will contain an animation.

If you're reading the book and you're near your Macintosh, use the In Action! program in conjunction with the book. When you get to a disk icon in the book you'll know that the In Action! program covers that topic. Figure 10 shows the disk icon in the book and the corresponding lesson in the In Action! program.

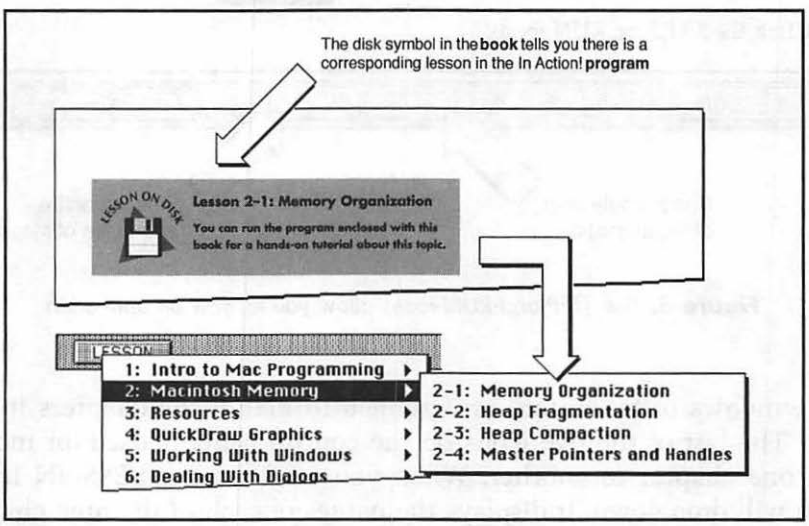
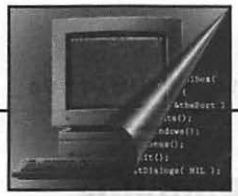


Figure 10. *The disk symbol in the book means there's a corresponding lesson in the software*

That's enough coverage of the preliminaries. Now, it's time to get down to work!



1 Introduction to Macintosh Programming

When you look at a Macintosh screen with the menus, windows, and icons that make up its graphical user interface, you discover that it's visually quite different from a PC or mainframe computer screen. The programming effort and techniques that go into achieving this effect are different as well.

If you currently program in a text-based system rather than a graphically-oriented one, this chapter will serve as your introduction to the differences between the two. If you program for MS Windows, you'll learn the similarities and differences between Windows and the Macintosh. And if you've programmed the Macintosh before, you'll get a refresher on Macintosh basics and perhaps gain a better understanding of the qualities unique to the Macintosh.

Development Systems

On a Macintosh, there are different means to accomplishing your programming goal. Besides using a programming language, you can also create a program using an information environment or an application framework.

Information environments

Every Macintosh comes with a program called HyperCard,which runs HyperCard stacks. These are programs written expressly for HyperCard and designed to display screens of information. A stack is not a standalone application. In order for users to run stacks, they must have HyperCard on their Macintoshes.

Although you can create simple stacks in a purely visual manner—that is, without any programming—most of the interesting stacks are written using HyperTalk, a language designed strictly for HyperCard. HyperTalk's strength is its simplicity, but it is also its weakness. To expand its usefulness, HyperTalk has the capability for adding functions written and compiled in a true programming language such as C or Pascal.

HyperCard's primary competition comes in the form of a Silicon Beach product called SuperCard. SuperCard is very similar to HyperCard, but its language is a little more powerful.

Application frameworks

An application framework is a sophisticated class library for object-oriented programming. A class library is a group of classes predefined for you. These classes provide the kinds of functionality needed by most programs, such opening and closing files, printing, and showing documents. The effect is to give you a functioning program shell. You write a minimal amount of code to turn this generic shell into a complete application that meets your needs.

Apple's MacApp and Symantec's Bedrock are examples of application frameworks. With a framework application like MacApp or Bedrock, you write the guts of a program. As an example, you program what goes into a window, then MacApp manages the window for you.

Programming languages

Most people who create programs for the Macintosh use a conventional programming language that allows them to write source code and then compile that code into a standalone application. You can buy a

Macintosh compiler for any of the major, and most of the less-than-major, programming languages. This includes Pascal, C, and C++.

This book assumes you will be using a programming language, rather than one of the information environments or an application framework mentioned above. Most of the example code provided in this book is in C, but all of the concepts and techniques are applicable to any higher-level language including C++ and Pascal.

About Macintosh Programming

The Macintosh has gained its enormous popularity with users because of its ease of use—its reputation as "the computer for the rest of us." For programmers, its reputation is altogether different. While its GUI, or *graphical user interface,* makes learning to *use* the Macintosh simple, it does nothing to make *programming* it easy. The "Macintosh way" presents a host of new challenges to programmers.

If you are a PC or mainframe programmer, be prepared to reorient yourself—completely.

If you are an MS Windows programmer you already know many of the programming concepts that will be new to others. But don't get too relaxed—Windows programming differs from Macintosh programming in many respects, and you'll still have much to learn.

If you've programmed the Macintosh, but aren't confident or satisfied with the level you are now at, it may be because you've pieced together your Macintosh applications without a sound knowledge of basic Macintosh programming techniques.

This book covers the fundamentals of Macintosh programming through in-depth discussions of general techniques and backs up that theory by providing many straightforward examples. You will receive a firm foundation on which you can build the Macintosh programs you want, regardless of the language you choose to program in.

Bit-mapped Graphics

The Macintosh, like other systems that use a GUI, uses bit-mapped graphics. Bit-mapped means that every *pixel*, or display dot, shown on the screen has a corresponding bit, or bits, in memory. The corresponding memory controls the status of each pixel. For a monochrome system, the memory keeps track of whether a pixel is on or off. For a color system, the memory keeps track of the color of each pixel. By way of contrast, in a character-mapped system a program cannot control pixels on the screen, it can control only text characters. Characters are located on a character grid, usually 25 rows by 80 columns.

In a bit-mapped system, each pixel is specified by a pair of coordinates that define a point, as in (20, 75). The first coordinate in the pairing describes the pixel's horizontal value; the second its vertical value. Pixel numbering begins at the upper-left corner of the screen, which corresponds to point (0, 0). Using this numbering system you can locate any pixel on the screen by giving its horizontal and vertical values.

To draw to the screen you must first specify a starting location, then perform the drawing operation. Here's an example:

```
MoveTo(30, 50);  /* move to pixel (30, 50)                          */
Line(0, 100);    /* draw a line downward, 100 pixels in length */
```

Unlike text-based systems, a bit-mapped system allows you to draw text anywhere on the screen. Note that I use the word draw when I speak of text. To the Macintosh, the distinction between creating text and drawing a shape is slight. In either case, the pixels are turned on to achieve the desired effect. Figure 1–1 shows both text and graphics and an enlarged view of the pixels used.

Figure 1–1 illustrates the advantage of using bit-mapped graphics—it's easy to mix text and graphics and place them anywhere on the screen.

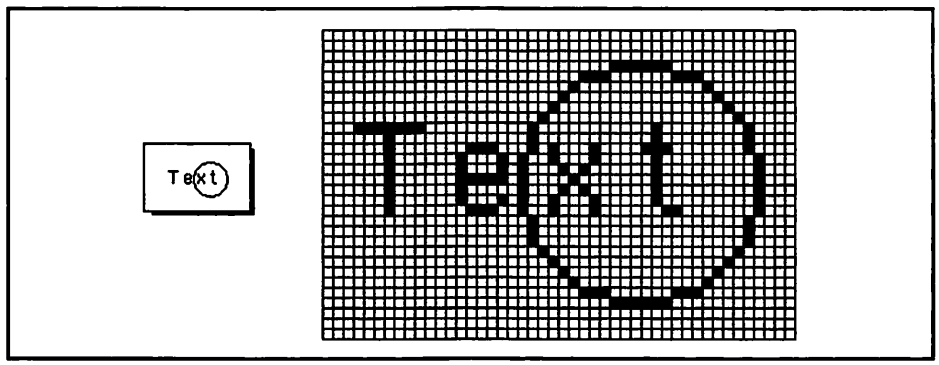


Figure 1–1. *Bit-mapped graphics*

Event-driven Programming

Programs that don't use a graphical user interface normally run in a sequential manner. Each time you run a program of this type you execute steps in the same order. For a program that displays four screens of information, like that shown in Figure 1–2, the program's user would generally view the four screens one after another in a predefined order.

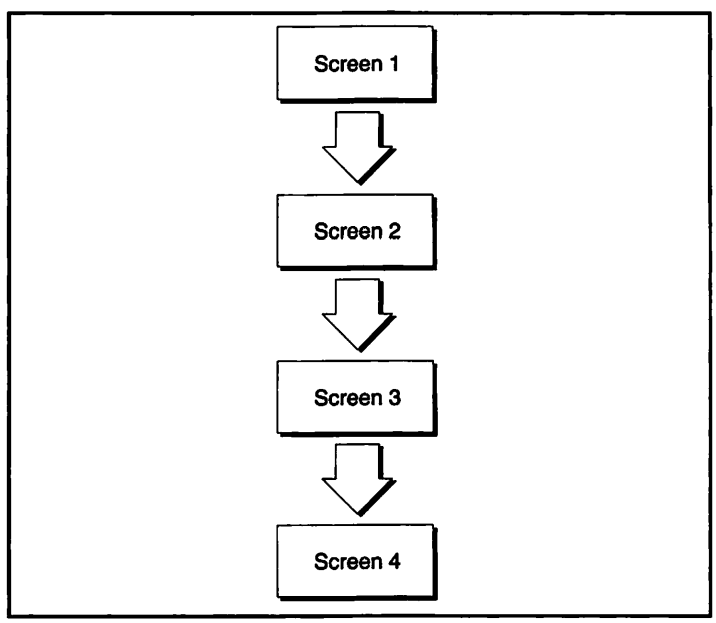


Figure 1–2. *Structure of a non-Macintosh program*

The key difference between these two types of programs is something Apple refers to as an *event*. A user's action, such as the press of a key or a click of the mouse button, produces an event. When an event occurs, the Macintosh system software automatically saves information about the event in an event record. The event record consists of fields that contain information about an event. If the event was a mouse click, the event's *what* field would then hold that information, that is, what type of event just occurred. The event record's *where* field would hold the screen location where the mouse click occurred.

Programs that use a GUI don't follow this linear pattern, nor are they limited to full screens to display their information. Instead, they use windows. The program's user is free to view the windows in any order. For a Macintosh, the window selection would most likely be based on a menu choice. The method used to make this selection is a keyboard or, more often, a pointing device such as a mouse. Figure 1–3 shows the structure of a Macintosh program.

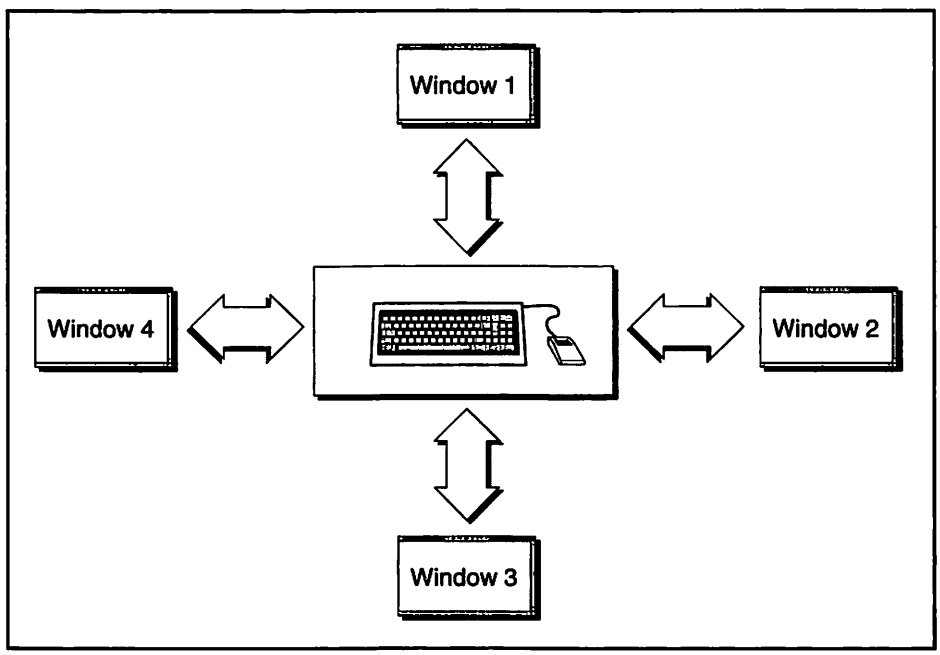


Figure 1–3. *Structure of a Macintosh program*

All Macintosh programs are controlled by a main loop. The purpose of this loop is to unceasingly retrieve and process events. As events occur they are stored in an *event queue*, which is serviced by the *event loop*. Here's a simple event loop:

```
EventRecord  The_Event;
Boolean      Not_Done = TRUE;

while (Not_Done)
{
   GetNextEvent(everyEvent, &The_Event);

   switch (The_Event.what)
   {
      case mouseDown:
         Handle_Mouse_Down();
         break;

      case keyDown:
         Handle_Keystroke();
   }
}
```

You use *GetNextEvent()* to retrieve a single event, storing the information in the event record variable *The_Event*. Then, based on the event type—the *what* field of the *EventRecord*—you process, or handle, the event. The above example reacts to two types of events: a mouse click and a keystroke. It responds to an event by calling the appropriate function that handles an event of that type—either *Handle_Mouse_Down()* or *Handle_Keystroke()*. You are responsible for writing these event-handling routines.

If you're an MS Windows programmer, retrieving events by calling *GetNextEvent()* from within a loop should sound very familiar to you. Windows programmers poll for messages by calling *GetMessage()* from within a loop. One big difference is that on a Macintosh there is a single event stream which all applications are aware of, while on Windows each window deals with its own designated message stream.

The accepted event-handling practice is as follows. Figure 1–4 illustrates the following steps.

- Use *GetNextEvent()* to retrieve an event to an event record.
- Use a *switch* statement to examine and respond to the event.
- Based on the event type, call a function to handle the event.
- Repeat the process.

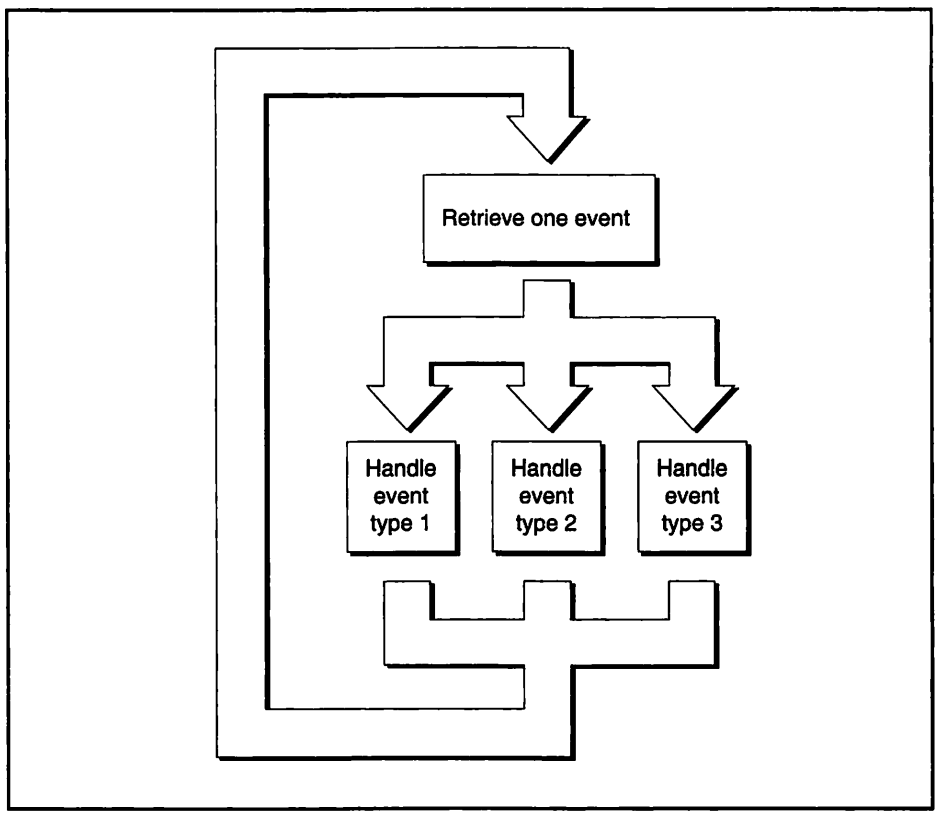


Figure 1–4. *Structure of an event-driven program*

For those of you who haven't programmed a graphical user interface, this discussion shouldn't be entirely foreign to you. You've still written programs that have a bit of this event-driven flavor to them. You may have written a program that displayed a menu on the screen, like the first one in Figure 1–5.

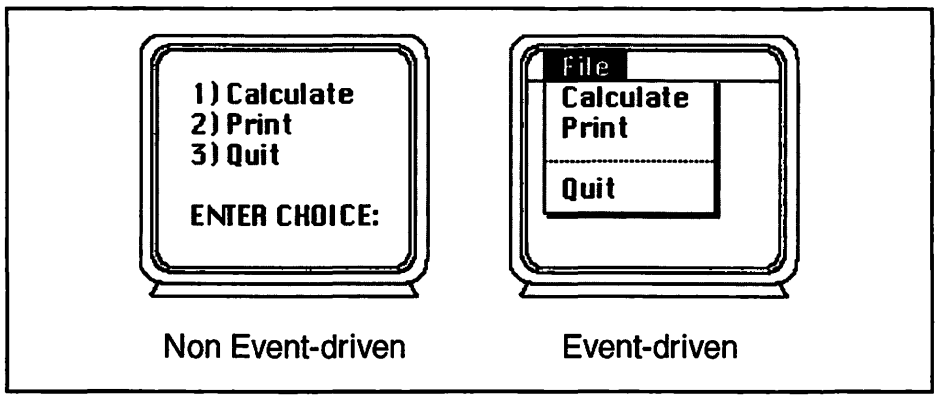


Figure 1–5. *Looping in non event-driven and event-driven programs*

Your program went into a loop, retrieving keyboard input using *scanf()* and then responding to this input. Here's an example:

```
Boolean  All_Done = FALSE;
int      choice;

while (All_Done == FALSE)
{
   scanf("%d", &choice);

   switch (choice)
   {
      case 1:
         Do_Calculations();
         break;
      case 2:
         Print_Results();
         break;
      case 3:
         All_Done = TRUE;
         break;
   }
}
```

The second graphic in Figure 1–5 shows how a Macintosh would display choices to the user. While the *scanf()* example waits for user input and then responds to it, it is not truly event-driven—it forces the user to wait at the screen until a choice is made from the limited menu.

The Macintosh, on the other hand, is aware of all types of events, such as keyboard input, mouse clicks, and the insertion of a disk into the computer. Most importantly, the user's actions control the type of event and the time the event will occur. This freedom and power that the Macintosh user enjoys are what makes events and the event loop such an important aspect of Macintosh programming.

Lesson 1-1: Events

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Resources

All of the things that make up a program's interface, such as menus, windows, dialogs, and icons, are called *resources*. Resources are not part of your source code, though your source code will be aware of them, use them, and eventually become linked to them.

It is advantageous to create a piece of the interface as a resource because a resource can be:

- Created and edited graphically, with no programming knowledge—even after a program has been published and distributed.
- Copied to another program for reuse.

For you MS Windows programmers, much of this should sound familiar. Macintosh resources and Windows resources are very similar. If you've only programmed for non-Windows PCs, or mainframes, pay close attention. In the Macintosh world, an appreciation for resources is very important.

To create a resource you use a resource editing program, such as Apple's ResEdit. You save a resource, or several resources, in a resource file. The icons for ResEdit and a ResEdit file are shown in Figure 1–6.

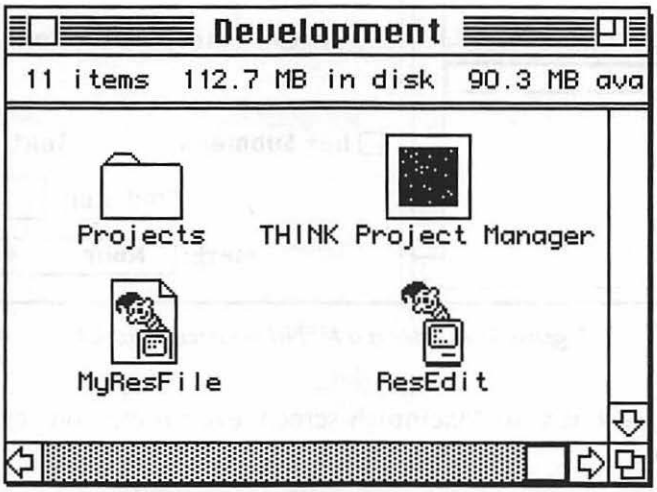


Figure 1–6. *The Icons of ResEdit and a ResEdit file*

Different components of a program's interface have different resource types. Each resource type has a four-letter, case-sensitive name. The resource type of a menu is 'MENU', for example.

A program such as ResEdit allows you to visually create a separate resource for each part of your application's interface. Figure 1–7 shows a 'MENU' resource being created. Instead of writing source code to define the items in a menu, you use ResEdit to create a 'MENU' resource.

MS Windows programmers may be familiar with editing resources using a tool such as Borland's Resource Workshop or Microsoft's AppStudio. ResEdit is Apple's version of a resource-editing program. As in Windows, resources can be created by purely visual means or by compiling a text representation of resources. Unlike Windows, most Macintosh programmers standardize on the visual method—ResEdit.

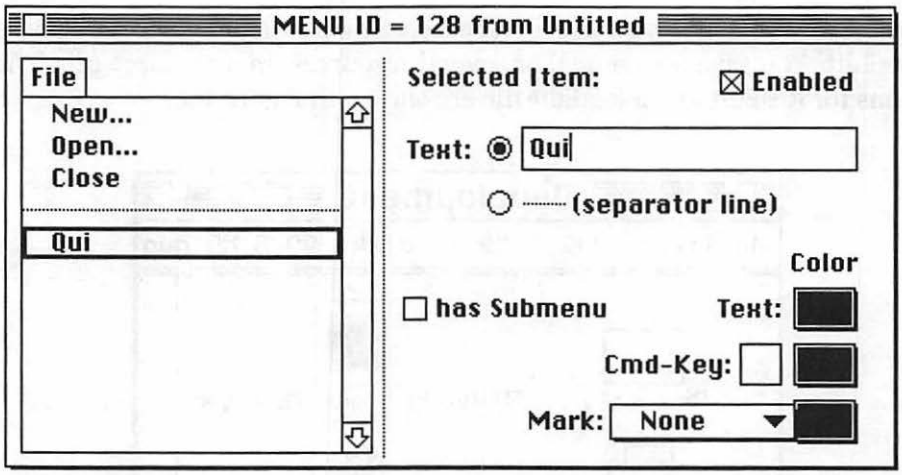


Figure 1–7. *Editing a MENU resource in ResEdit*

When you look at your Macintosh screen, everything you see originated as a resource:

- A menu bar has an 'MBAR' resource that specifies which individual menus are in it.
- Each individual menu has its own 'MENU' resource that defines the items in that menu.
- A window has a 'WIND' resource that defines its size and initial position on the screen.
- A dialog has a 'DLOG' resource that defines its size and initial position.
- A dialog has a second resource, the 'DITL', that defines items such as buttons that are to appear in the dialog.

Figure 1–8 illustrates this.

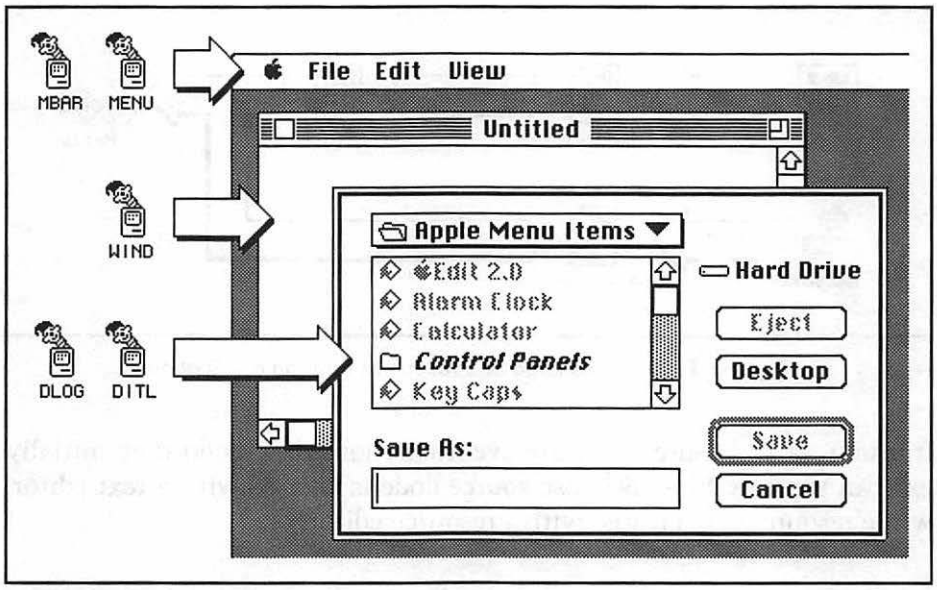


Figure 1–8. *Everything that you see on your Macintosh screen has a resource that defines it*

Once you've created the resources that define the screen elements of your program, you write source code that uses these resources. I'll have a lot more to say about the source code/resource connection throughout this book.

The THINK C compiler, like most compilers, includes a source code editor, compiler, and linker within its one environment. When it is time to turn your source code into a standalone application THINK C compiles your source code, then joins the compiled code with the resources in your resource file. The result is an application. This process is shown in Figure 1–9.

In a development environment like THINK C, you will not actually see a file such as the *Hello.o* file shown in Figure 1–9. THINK C holds all object code in something it calls a project file. Since your attention will be directed towards the source code and the final application, the fact that object files are invisible to you should not be a concern.

When you link a Macintosh program, the linker converts the object code into 'CODE' resources and stores them, along with the resources you created earlier, in the final application. In the end, a Macintosh program is almost completely composed of resources.

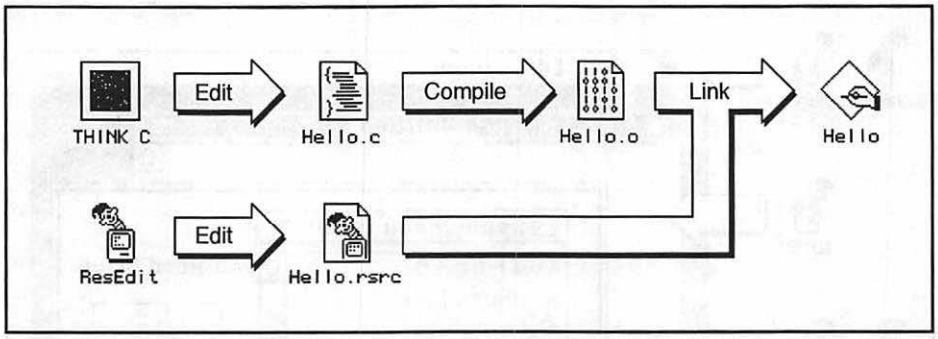
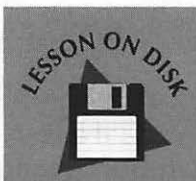


Figure 1-9. *Source code and resources form an application*

If resources and source code are eventually joined, why do they initially exist in separate files? Because source code is created with a text editor, while resources are created with a resource editor.



Lesson 1-2: Resources

You can run the program enclosed with this book for a hands-on tutorial about this topic.

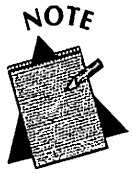
The Toolbox

With a resource editing tool such as ResEdit, creating menus, windows, and dialogs is easy. But a resource contains only a description of a piece of the interface—it doesn't do anything with it. For example, ResEdit easily allows you to list the items that will be in a menu. To then display that menu—to track the user's mouse movements over it and then drop it down and display the items in it—you need to write source code.

The menu scenario just described shows up in every Macintosh program. You can therefore infer that much of the code to perform that scenario should look the same in any Macintosh program. The phrase "don't reinvent the wheel" comes up a lot in programming, and Macintosh takes this phrase to its limit. Apple programmers wrote a few thousand routines that handle all of the actions common to most

Macintosh programs. They then graciously gave them away—free. Well, not exactly free. To get the thousands of routines, you have to buy a Macintosh computer.

Instead of creating libraries of routines, as is the common practice with languages for other computers, Apple has taken the code that makes up these routines and burned it into ROM chips that are then placed inside each Macintosh. Collectively, Apple refers to these routines as the *Macintosh User Interface Toolbox*, or *Toolbox* for short.



If you're a PC or mainframe programmer who has never programmed in a windowed environment, don't let the idea of these invisible routines overwhelm you. On a mainframe or PC you also use routines that you didn't write, like standard C library functions such as *strlen()* and *printf()*. You just don't have a fancy name for them like the Macintosh User Interface Toolbox!

Figure 1–10 shows how we'll illustrate the Toolbox emphasizing the point that the code for Toolbox routines lies in ROM and not in your source code.

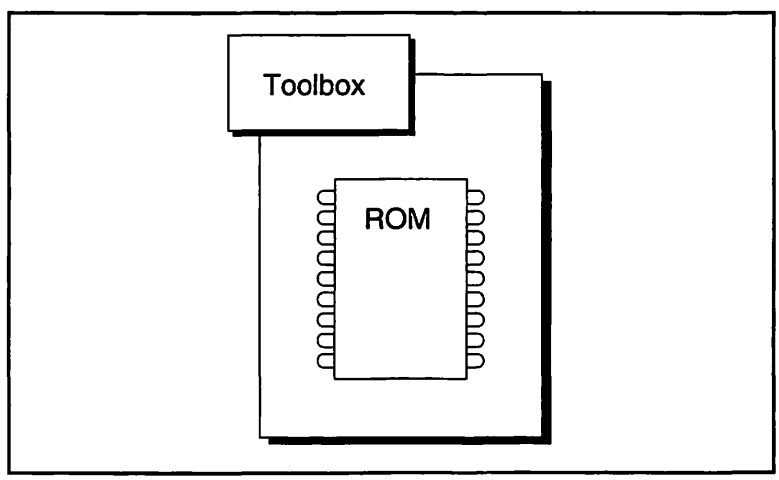


Figure 1–10. *The Toolbox is in ROM*

PCs have software built into their ROMs too—the ROM BIOS services. The difference? The Macintosh Toolbox is easier to use and provides a means to display and work with graphics and a sophisticated user interface.

In the previous section I said that you first create a resource and then write source code that uses that resource. This, of course, implies that your source code somehow communicates with your resources. Toolbox routines are this communication link. Here's a brief example:

```
WindowPtr  the_window;
the_window = GetNewWindow(128, 0L, (WindowPtr)-1L);
```

The routine *GetNewWindow()* is a Toolbox function that locates a 'WIND' resource and loads it into memory. The code that makes up *GetNewWindow()* exists in ROM. When your source code makes a call to *GetNewWindow()* your program is interrupted while the code in ROM is executed.

A call to *GetNewWindow()* gives your program a pointer to the window—a *WindowPtr*. The *WindowPtr* variable is your means of identifying this one particular window. When you perform operations on the window, such as moving it or drawing graphics to it, you'll use the window's pointer to refer to the window. There will be more on this topic in Chapter 5.

The *GetNewWindow()* code in ROM looks to a 'WIND' resource in your program's resources to determine the type and dimensions of the window. Notice in the above example that there are parameters passed to *GetNewWindow()*. The first parameter is an ID that tells the Toolbox which 'WIND' resource to use because your resource file may hold more than one 'WIND' resource. The Toolbox routine looks to the resource file for information about the window, and the resource file passes this information back to the Toolbox.

Figure 1–11 summarizes the relationship between source code, the Toolbox, and resources. In this figure a call to *GetNewWindow()* accesses the Toolbox, which in turn looks at the program's resources to get the descriptive window information it needs to put up the correct type and properly-sized window. This information is contained in a 'WIND' resource.

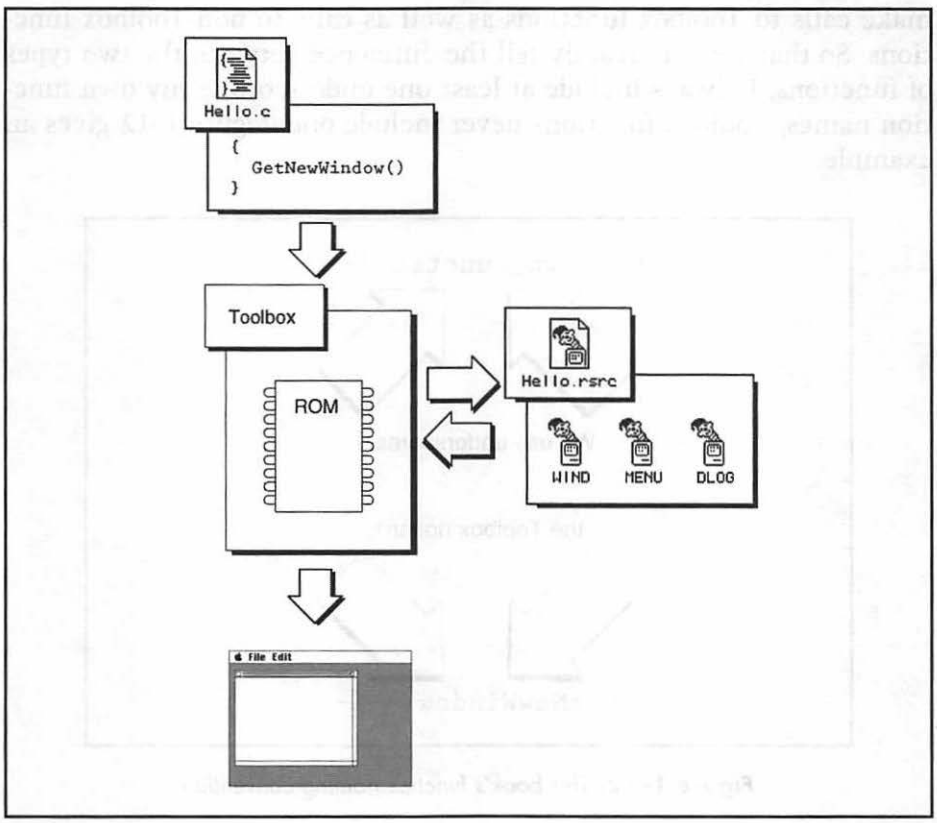


Figure 1–11. *Source code, resources, and the Toolbox*

The *GetNewWindow()* routine is just one of the thousands of functions in the Toolbox. To find the ones that serve your purposes, and to know the parameters to pass, refer to Appendix D of this book. There we list many of the more commonly-used Toolbox routines. For the complete listing of Toolbox routines you'll have to turn to Apple's *Inside Macintosh* series of reference books. The books in this library are reference books, not beginner-level tutorials.

For MS Windows programmers, the *Inside Macintosh* reference books are the Macintosh counterpart to your *Windows Programmer's Reference* manual.

This book covers many of the Toolbox functions. The examples often make calls to Toolbox functions as well as calls to non-Toolbox functions. So that you can readily tell the difference between the two types of functions, I always include at least one underscore in my own function names, Toolbox functions never include one. Figure 1–12 gives an example.

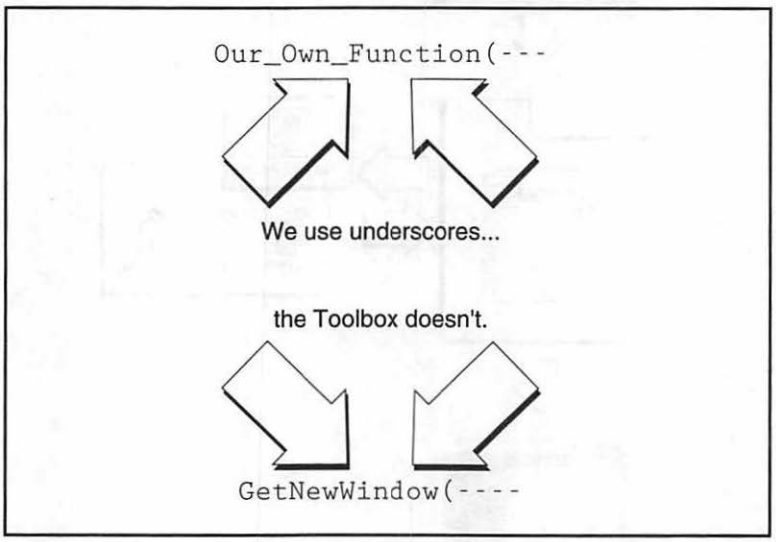


Figure 1–12. *This book's function naming convention*

Lesson 1–3: The Toolbox

You can run the program enclosed with this book for a hands-on tutorial about this topic.

The Operating System

Like the Toolbox, the code that makes up the Macintosh *Operating System* is located in ROM—that's why Figure 1–13 is so similar to Figure 1–10. The Operating System is different from the System file, which is found in the System Folder, and is described in the next section.

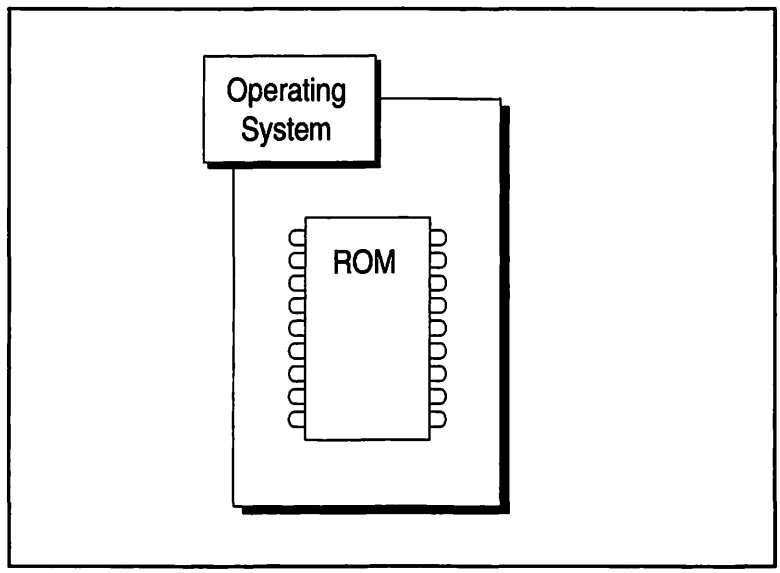


Figure 1–13. *The Operating System is in ROM*

The Operating System, like the Toolbox, consists of routines that you can access by way of function calls in your source code. The difference between the routines in the Toolbox and those of the Operating System is in the level of the tasks they perform. Operating System routines deal with low-level tasks such as handling keystrokes and disk-insertions. Toolbox routines deal with higher-level tasks. The result of a higher-level chore is more noticeable to the user, such as the display of windows and the drawing of shapes or pictures in those windows.

You perform an Operating System task just as you do a Toolbox task—you make the appropriate function call. An example of an Operating System call is *Eject()*, which physically ejects a disk from the floppy disk drive.

PC and mainframe programmers will appreciate the simplicity of accessing the Macintosh Operating System. To perform a task you need only know the proper Operating System routine to call; you use no direct-memory addressing using jumps or interrupts.

System Software

Now that you know what Toolbox routines and Operating System routines are, you can refer to them collectively as *system software.*

System software is divided into the two broad categories of the Toolbox and the Operating System. It is then further sectioned into groups of functionally-related routines. These groups are called *managers.*

The Window Manager is an example of a manager. It consists of routines such as *GetNewWindow()* and *MoveWindow()*—routines that allow you to create and work with windows. Some of the other managers are given in Figure 1–14.

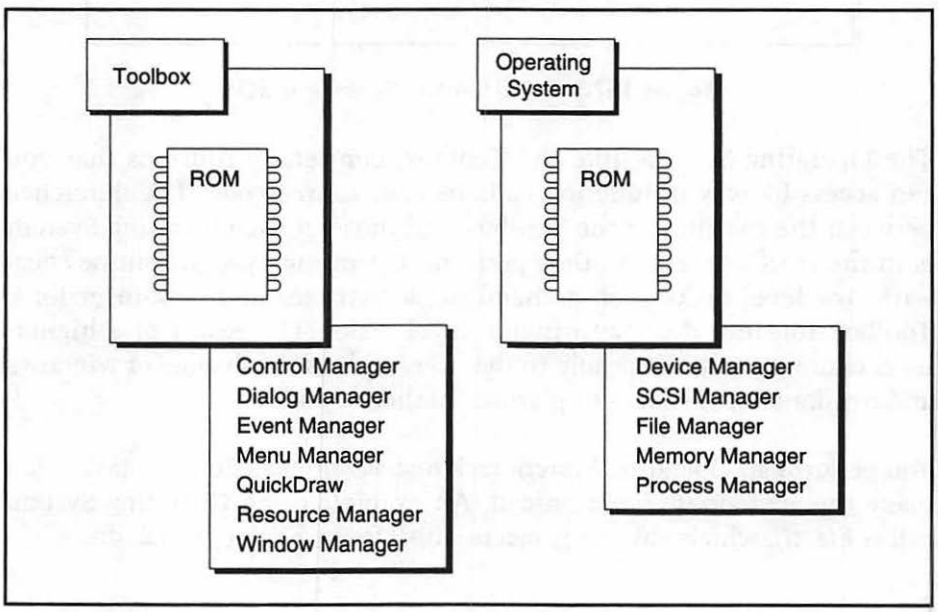


Figure 1–14. *The Macintosh managers*

From the names of the managers in Figure 1–14 you can see that the Toolbox managers deal with the user interface: windows, dialogs, and menus. The managers that comprise the Operating System, on the other hand, deal with low-level tasks such as memory management and the control of devices such as monitors.

I'll have more to say about individual managers throughout this book.

The System File and Finder

The System Folder that appears on every Macintosh contains two files of note: the System file and the Finder.

The System file

The *System file*, not to be confused with the Operating System in ROM, consists of resources that are accessible by all programs. New models of the Mac may contain new versions of the ROM and thus new, previously unavailable system software. An owner of an older Macintosh can access these new Toolbox and Operating System routines by upgrading to a new System file.

The System file contains *patches*—code that takes the place of the routines found in new versions of ROM. When an older version of ROM receives a call to a new system routine, it can look to the System file and run the code from there. Figure 1–15 illustrates this idea. The figure shows that with the new ROM the Toolbox can handle the new routine directly. The older ROM must access the new System file to make use of the new routine.

If a Macintosh user has an old version of ROM and an older System file, an application attempting to make a call to a new routine will crash. With the multitude of Macintosh models in use, this is a significant problem. I introduce techniques to avoid this type of crash in Chapter 8.

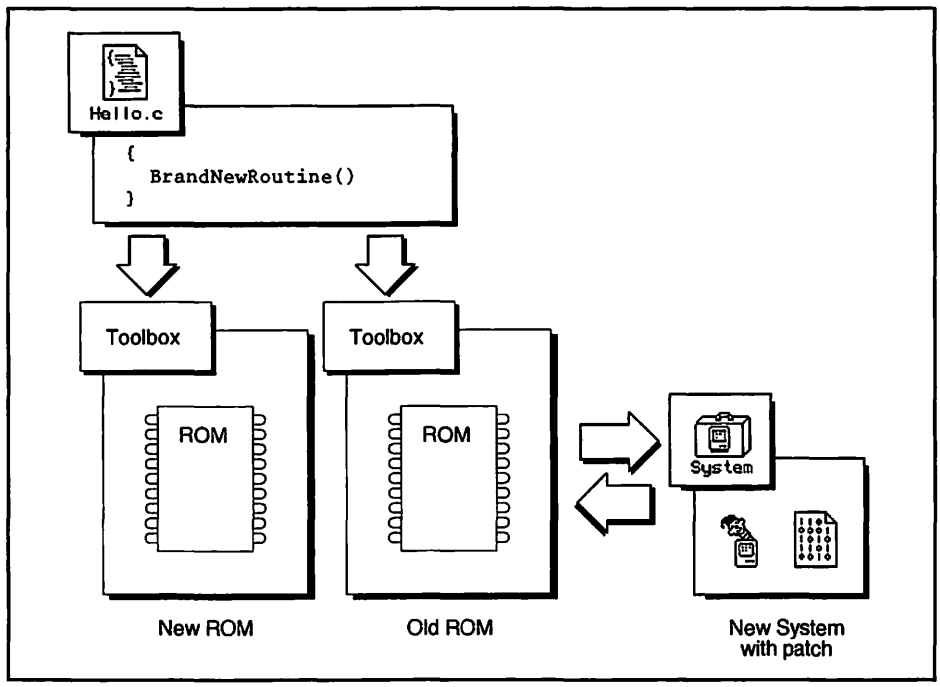


Figure 1–15. *Patches take the role of ROM routines*

The Finder

The *Finder* is a program that, like most Macintosh programs, consists of code and resources. The Finder is loaded into memory and starts running when you turn on your computer. It is responsible for displaying the desktop pattern and the icons you see on it, such as the trash can, files, and folders. When you move, copy, and delete files the Finder is doing the work. The Finder makes use of some of the common resources in the System file to display the interface that the user sees. Figure 1–16 shows the System file and the Finder and what the Finder is responsible for doing.

On PCs running DOS, there is no real equivalent to the Finder and to the base-level user interface it provides. Unless you consider the "C:>" prompt in DOS to be a user interface!

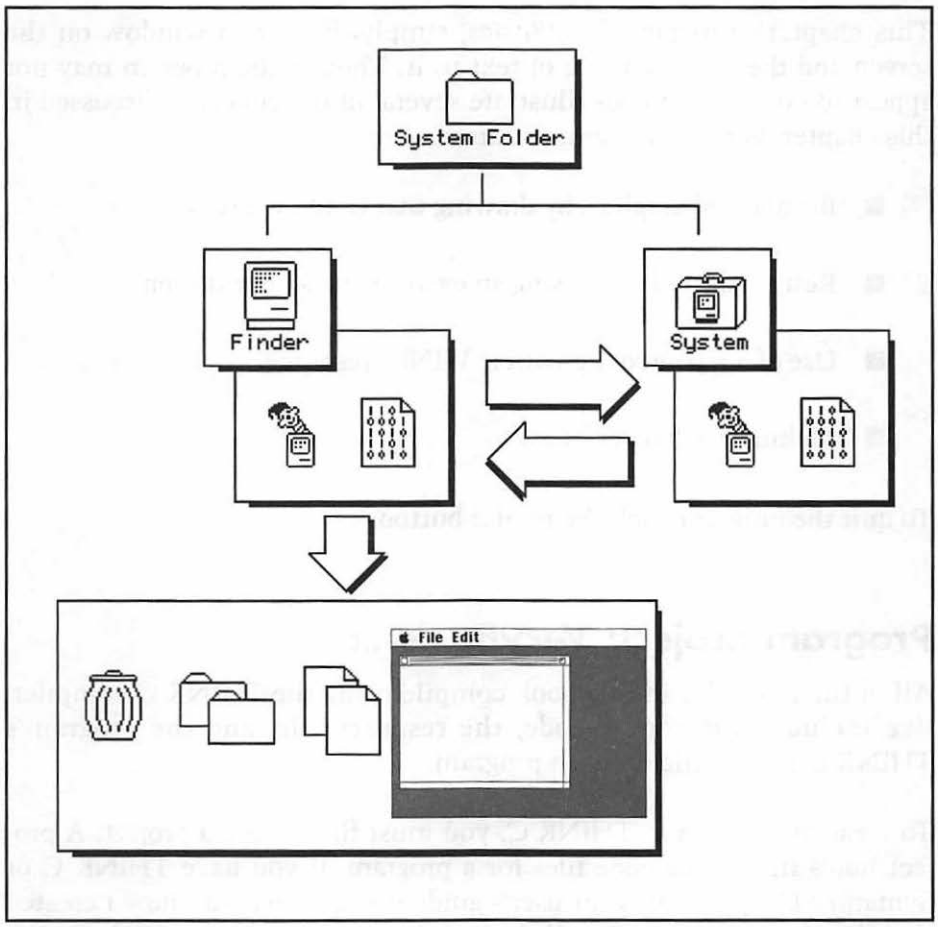


Figure 1-16. *The Finder and the System file*

Chapter Program: Intro to Mac Programming

We'll close this chapter, and every chapter hereafter, with a short sample program that demonstrates the chapter's topics. The source code for each chapter is included on the disk that came with this book. If you own a C or C++ compiler, such as THINK C or Symantec C++, you can compile and run any of the programs.

This chapter's program, *VeryBasics*, simply displays a window on the screen and then draws a line of text to it. Though the program may not appear to do much, it does illustrate several of the concepts discussed in this chapter. *VeryBasics* demonstrates:

- Bit-mapped graphics by drawing text to the window.
- Retrieving and processing an event using an event loop.
- Use of a resource file with a 'WIND' resource.
- Making Toolbox calls.

To quit the program click the mouse button.

Program project: *VeryBasics.π*

All of the examples in this book compile using the THINK C compiler. I've included the source code, the resource file, and the program's THINK C project file for each program.

To create a program in THINK C, you must first create a project. A project holds the source code files for a program. If you have THINK C or Symantec C++, refer to your user's guide if you aren't sure how I created the *VeryBasics* project. I will, however, tell you a little bit about the THINK C file naming convention.

A project typically has the same name as the executable program but also has a *.π* extension. The π character is created by pressing the Option key and the letter "p" key. Thus, the *VeryBasics* program has a project named *VeryBasics.π*.

When it is time to compile the program, THINK C will look for a resource file with the project name plus the extension *.rsrc*. The *VeryBasics* resource file, covered in the next section, has the name *VeryBasics.π.rsrc*.

Program resources: *VeryBasics.π.rsrc*

For the first resource file, I'll quickly run through the steps used to create the file and the one and only resource the *VeryBasics* program uses.

1. Run ResEdit.
2. Click on the introductory dialog to dismiss that dialog.
3. Click the New button in the next dialog that opens.
4. Name the resource file VeryBasics.π.rsrc.
5. Click the New button.
6. Choose Create New Resource from the Resource menu.
7. Scroll to the 'WIND' type, then click on it.
8. Click the OK button.
9. You now have a 'WIND' resource. If you wish, click on one of the small window icons to change the type of window. Type in new values in the four size edit boxes to change the dimensions of the window.
10. Choose Save... from the File menu.
11. Choose Quit from the File menu.

Figure 1–17 shows part of the editing window for the 'WIND' resource.

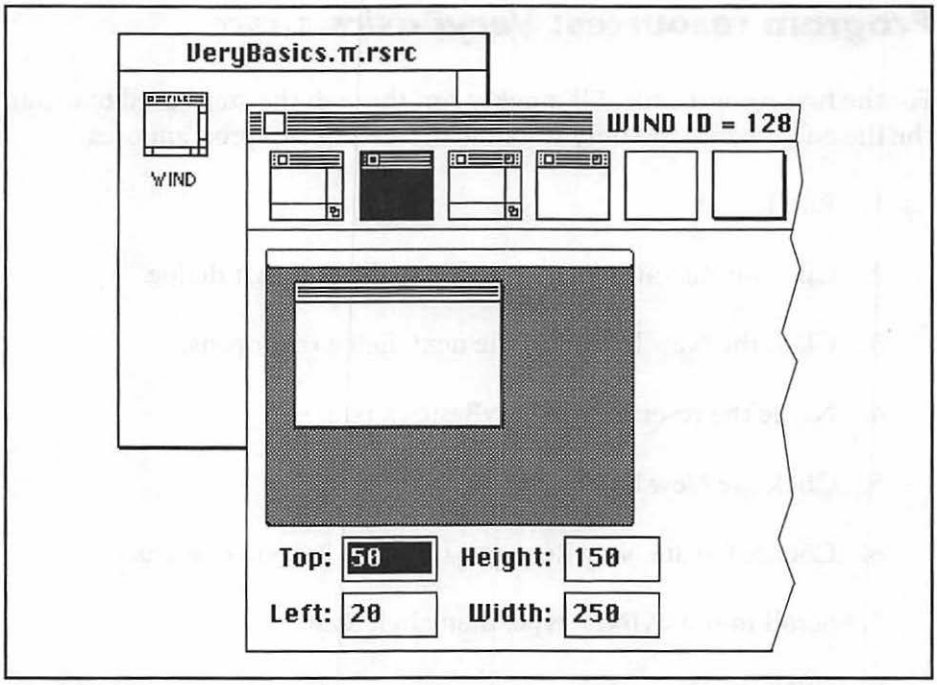


Figure 1-17. *Creating a 'WIND' resource in ResEdit*

Program listing: *VeryBasics.c*

In the sections following the program listing I describe the key elements of the source code.

```
/*++++++++++++++++++    Function prototypes    ++++++++++++++++++*/

void  Handle_Mouse_Down( void );


/*++++++++++++    Define global constants    +++++++++++++++++++*/

#define    WIND_ID                    128
#define    BEEP_DURATION                1
#define    NIL                         0L
#define    IN_FRONT       (WindowPtr)-1L
#define    REMOVE_EVENTS                0
```

```
/*+++++++++++++++   Define global variables   +++++++++++++++++*/

WindowPtr     The_Window;
Boolean       All_Done = FALSE;
EventRecord   The_Event;


/*++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();

   The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

   SetPort( The_Window );

   MoveTo( 30,50 );
   DrawString( "\pChapter One Program" );

   while ( All_Done == FALSE )
   {
      GetNextEvent( everyEvent, &The_Event );

      switch ( The_Event.what )
      {
         case mouseDown:
            Handle_Mouse_Down();
            break;
      }
   }
}


/*++++++++++  Handle a click of the mouse button  ++++++++++*/

void  Handle_Mouse_Down( void )
```

```
{
    SysBeep( 5 );
    All_Done = TRUE;
}
```

Stepping through the code

If you're new to Mac programming, there are several lines of code in the listing that will look unfamiliar to you. Let's examine them here.

Where are the #includes?

When you look at the C source code for programs that run on non-Macintosh systems, the first thing you usually see are several *#include* directives that include header files in the program. Macintosh programs also use *#includes*, but you usually need to include only one header file, and that's done for you automatically.

Your C or C++ compiler gets its information about the calling convention of a Toolbox routine from a header file. Macintosh compilers come with approximately 100 header files. When you make a call to a Toolbox routine such as *GetNewWindow()*, your compiler looks to the *Windows.h* header file to find the prototype for *GetNewWindow()*.

The THINK C compiler has taken about 30 of the most commonly used header files and precompiled them into one header file called *MacHeaders*. Symantec C++ has a similar header file called *MacHeaders++*. The THINK Project Manager automatically includes this *MacHeaders* in every source code file you use, so you don't have to use *#include* directives for it or for any of the included files it uses.

MS Windows programmers know that Windows programs always include one large header file—*Windows.h*. If a Windows compiler automatically included this header in all of its windows source code, it would function in the same way as the THINK C compiler that includes *MacHeaders*.

There will be times when you need to include some of the other Macintosh header files in a program. When you do, you simply use standard *#include* directives. Examples abound throughout this book. Like any other compiler, Macintosh compilers also allow you to write your own headers and include them.

Function prototypes

Prototypes aid the compiler in determining if functions are being called properly. Though some compilers might let you slip by without them, always use them. For the Macintosh, prototypes are written in the same form as they are for any other computer whose compiler supports this construct. You do not have to include a variable name when you list the arguments, just the type of the argument. Here's the prototype of the *Handle_One_Event()* function:

```
void  Handle_Mouse_Down( void ):
```

If you program on an older minicomputer or mainframe, it is possible that your C compiler does not support prototypes—a relatively recent extension to the C language. If so, consult any book that describes the ANSI standard definition of the C language.

The #define directives

As you can tell from the listing, Macintosh programs use *#define* directives in the same manner as *#defines* are used by compilers for other computer systems. I'll comment on each *#define* as it appears in the listing. Here are the five *#defines VeryBasics uses*:

```
#define    WIND_ID                    128
#define    BEEP_DURATION                1
#define    NIL                         0L
#define    IN_FRONT        (WindowPtr)-1L
#define    REMOVE_EVENTS                0
```

Global variables

Variable declarations take on the same format for Macintosh C as they do for other versions of C. Macintosh C, however, has some data types all its own. I'll cover many of these types at various places in this book, and we summarize them in Appendix A. Here are *VeryBasics'* three global variables:

```
WindowPtr     The_Window;
Boolean       All_Done = FALSE;
EventRecord   The_Event;
```

The main() function

```
void  main( void )
{
   ...
   ...
}
```

Like other C programs, Macintosh programs always begin at the *main()* function. And like all C programs, you don't explicitly call *main()*; it is automatically the first function to execute when you run a Macintosh program.

MS Windows programs use *WinMain()* rather than *main()*.

Toolbox initialization

The various managers must be initialized before performing calls to Toolbox routines. The sequence of initialization calls given here should be included in every Macintosh program you write, in the order given here:

```
InitGraf( &thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( NIL );
FlushEvents( everyEvent, REMOVE_EVENTS );
InitCursor();
```

Two of the constants I defined are used here. On the Macintosh, pointers occupy four bytes of memory. A nil pointer has a value of 0. By defining *NIL* to be *0L* I force the value zero to occupy the size of a long variable of four bytes.

The *FlushEvents()* routine clears extraneous events from the event queue in preparation for the start of the program. The second parameter to *FlushEvents()*, the *REMOVE_EVENTS* constant, specifies that all events in the queue should be wiped out.

A call to a Toolbox routine that exists in a manager that was not initialized will crash your program.

Loading a window

Finally, some action! A call to *GetNewWindow()* loads a 'WIND' resource into memory. When you create a 'WIND' resource in ResEdit, you have the option of specifying whether the window should be visible or hidden when this call is made. If you examine the 'WIND' resource in the *VeryBasics.π.rsrc* file you'll see that the check box labeled "Initially visible" is checked. That means that when this call is complete a window will appear on the screen. Here's the call that loads the 'WIND' resource:

```
The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
```

In a Mac program there may be more than one occurrence of a resource type. In order to be able to specify which resource you want to use, every

resource has an ID. When *VeryBasics* calls *GetNewWindow()*, the ID of the 'WIND' to load into memory is the first parameter. I gave WIND_ID the value of the resource ID of the 'WIND'.

The second parameter to *GetNewWindow()* tells the Window Manager where in memory to store this newly loaded window. Using a nil pointer here tells the Window Manager to use whatever available memory it wants. Chapter 5 shows you how to be more specific about where in memory the window should be stored.

The last parameter to *GetNewWindow()* specifies whether the new window should open in front of or behind all other open windows. This is the program's only window, so this parameter doesn't have an impact on the call. In general, you'll open a new window in front of all others. A value of -1 accomplishes this. Also, this value must be a pointer and by convention it is -1L. And, more particularly, this value must be a special type of pointer: a window pointer. Preceding the value with (*WindowPtr*) casts the value -1L to a *WindowPtr* type. I've defined *IN_FRONT* to have this value of (*WindowPtr*)-1L.

If you are a PC programmer or write code for a machine in which pointers and integers are not the same size, you'll notice that Macintosh programmers are much more relaxed about placing integer values such as -1L and 0L in slots meant for pointers. Since they're both 32 bits in size, it all works out.

Drawing to a window

Every window has its own drawing environment, that is, its own port. That's how different windows can do things like display text in fonts different from one another. Before drawing to a window, you must set the port to that window. A call to the Toolbox routine *SetPort()* accomplishes this. The parameter to *SetPort()* is a pointer to the window whose port you want to use.

To move to a particular area in a window, you use the Toolbox routine *MoveTo()*. The first parameter is the horizontal location to move to, the second parameter is the vertical position. The effect of *MoveTo(30,50)* is

as follows: Start at the window's upper-left corner. Move 30 pixels to the right. Move 50 pixels down. Stay put until asked to move again or until asked to draw.

The Toolbox routine *DrawString()* draws a single line of text to a window. The line of text is preceded by *\p*, and the entire string is placed in double quotes. The Toolbox will be looking for a string in Pascal format. Strings which are in Pascal format are not terminated with a null byte, as they are when in C format. Rather, Pascal strings begin with a byte that contains the size of the string, followed by the text bytes of the string. Since our string is sent to it in C format, we let the Toolbox know this so that it can make the internal conversions necessary to display the string.

PC programmers use the \ character all the time:

```
printf("Start a new line.\n");
```

It should make sense that the escape character \ is used to signal the compiler that the letter *p* that follows does not stand for the letter in the alphabet, but rather indicates that the string that follows is in Pascal format.

Here's the code that sets the port, moves to the right and down, and then draws a line of text:

```
SetPort( The_Window );
MoveTo( 30,50 );
DrawString( "\pChapter One Program" );
```

The main event loop

The event loop, the driving force of the program, appears just as discussed earlier in the chapter. The only event type *VeryBasics* handles is a click of the mouse. It handles this *mouseDown* event by calling, or invoking, a procedure. Here it is:

```
while (All_Done == FALSE)
{
   GetNextEvent(everyEvent, &The_Event);

   switch (The_Event.what)
   {
      case mouseDown:
         Handle_Mouse_Down();
         break;
   }
}
```

A Macintosh function

```
void  Handle_Mouse_Down( void )
{
   SysBeep(5);
   All_Done = TRUE;
}
```

The Macintosh requires several new ways of orienting the process of writing a computer program. So you'll be happy to learn that the Macintosh does some things exactly as you've done in the past! A function for a Macintosh program is written and invoked in the same manner as a function you write for any other computer.

The *Handle_Mouse_Down()* routine simply beeps the Macintosh speaker with a call to the Toolbox routine *SysBeep()*. How do you know that *SysBeep()* is a Toolbox routine and *Handle_Mouse_Down()* isn't? Remember our naming convention. *SysBeep()* has no underscores in its name—it's a Toolbox routine. *Handle_Mouse_Down()* has underscores, which means that I wrote this routine.

Finally, *All_Done* is set to true. When the event loop attempts to execute again, the test of *All_Done* for false will fail and the program will end.

Chapter Summary

The Macintosh graphical user interface, or GUI, presents special challenges to programmers of the Macintosh. This book presents the techniques to overcome these challenges.

The Macintosh uses bit-mapped graphics. You can turn each pixel, or display dot, on or off on the screen. Each pixel has a pair of coordinates that make up a point that defines its position on the screen.

Macintosh programs don't run in a sequential, linear manner. Instead, a Mac program responds to events—user actions such as a click of the mouse button. An event record holds descriptive information about a single event. A Macintosh program is driven by an event loop-code that repeatedly checks for and responds to these events.

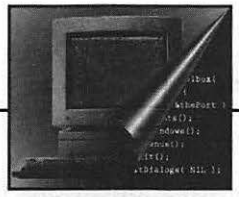
All elements of a Macintosh program, such as its menu, windows, and dialog boxes, are resources. A resource is a description of one of these elements. A 'WIND' resource, for example, holds the type, or look, of a window. It also defines the size of the window and the screen location where it will first appear. Resources can be graphically, or visually, edited using a program such as Apple's ResEdit resource editor.

Resources are simply descriptions of interface elements; they don't do anything with the elements. For that, you must write source code. So that you don't have to start from scratch, Apple provides thousands of prewritten functions to help you in working with resources. These routines are stored in the ROM of your Macintosh and are collectively referred to as the Toolbox.

The Macintosh Operating System, like the Toolbox, consists of routines you access from within your source code. The Operating System routines are low-level functions that perform tasks such as handling keystrokes, while the Toolbox routines are higher-level, performing the more noticeable tasks such as displaying windows and drawing pictures.

Collectively, the Toolbox and Operating System are called system software. The system software is divided into groups of functionally-related routines—managers. The Window Manager and Menu Manager are two examples.

The System File, found in the System Folder of each Macintosh, contains resources that are shared by programs. The Finder is another program found in the System Folder. It starts executing when your Macintosh starts up. It is responsible for displaying the desktop pattern and for performing file housekeeping like copying and deleting files.



2 Macintosh Memory

Understanding how the Macintosh works with memory is an important and often understudied topic. A knowledge of what is going on in RAM will aid you in writing programs that behave in a predictable manner.

The Macintosh uses a set of terminology and concepts all its own. This chapter will make you familiar with the basic terms and techniques of Macintosh memory. In Chapter 9 you will discover the details of memory management and learn actual techniques you can use to avoid memory problems.

In this chapter you will learn how memory is organized into partitions. You'll see how each partition is composed of the same basic areas of memory. You will also learn the techniques the Macintosh uses to make the most efficient use of memory.

Memory Organization

The Macintosh Operating System divides RAM into two main sections, or *partitions*. The partition at the low end of memory is the *system partition* and is reserved by the Macintosh for its own use. The Macintosh

dedicates the other partition to applications that you run. The Mac will further subdivide this partition into *application partitions*. For every application you run there is a corresponding application partition. Figure 2–1 illustrates this.

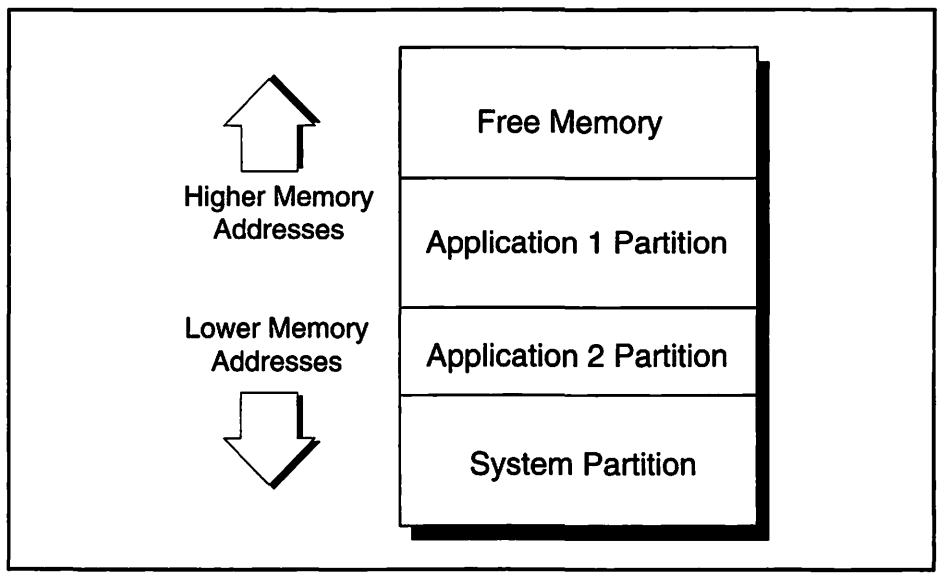


Figure 2–1. *Memory organization*

What does a RAM partition contain? That depends on whether the partition is a system partition or an application partition. Figure 2–2 shows RAM when a single application is running. I'll refer to this figure when I first describe the memory organization of the system partition. Then I'll refer to the application partition.

System partition organization

The RAM of a Macintosh always contains a single *system partition*. This is true regardless of the number of applications that may be running. The system partition has a section that contains system global variables and a section called the system heap.

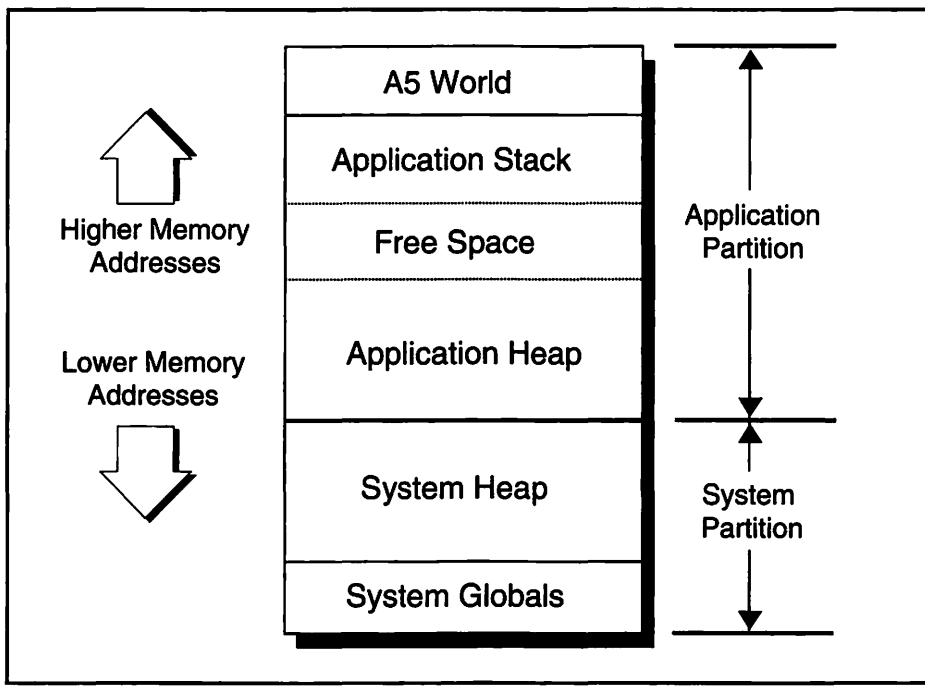


Figure 2–2. *The system and application partitions*

System global variables

At the bottom of memory, starting at address 0, the Mac reserves a section of memory for *system global variables*. The operating system uses these variables to keep track of what is going on in the operating environment. There are also variables stored here that establish constant environment values, such as the pixel height of the menu bar. Seldom, if ever, will it be necessary for you to directly use these variables. Instead, you will access them indirectly through Operating System and Toolbox routines.

System heap

Above the system global variables is the *system heap*. Only the Operating System uses this section of memory; you will never have a need to access information contained within it. The system heap contains things such as system file resources, extensions, and the code necessary to run the Finder. When you start up the Macintosh, the system

heap size is set and remains fixed until the next time the computer starts. At startup, *extensions* (such as Apple's QuickTime) call upon a software mechanism to expand the system heap to accommodate them. That's why you have to restart your computer once you move an extension into your system folder.

Application partition organization

When a program launches, the operating system reserves a section of free RAM for that application's use. This *application partition* devotes itself entirely to that application for the duration of the application's execution. When you quit that application the memory within that partition becomes free for the Macintosh to use for a different application.

A system partition has an *A5 World* that holds application global variables, an *application stack* that holds application local variables, and an *application heap* section that contains the program's code and resources.

A5 World

A program's global variables are stored in a section of the application partition called the A5 World. The name "A5 World" comes from the fact that the operating system uses the CPU's A5 register to keep track of just where this memory section starts.

Variables stored in the A5 World are accessible only to the program in this application partition. On the other hand, variables in the system partition are accessible by both the system and any application that is executing. Figure 2–3 illustrates this.

Application stack

The A5 World section holds variables global to the program in the application partition. The *application stack* is a section of memory used for holding the local variables of the program to which the application partition is dedicated. The stack also holds parameters as they are passed to functions.

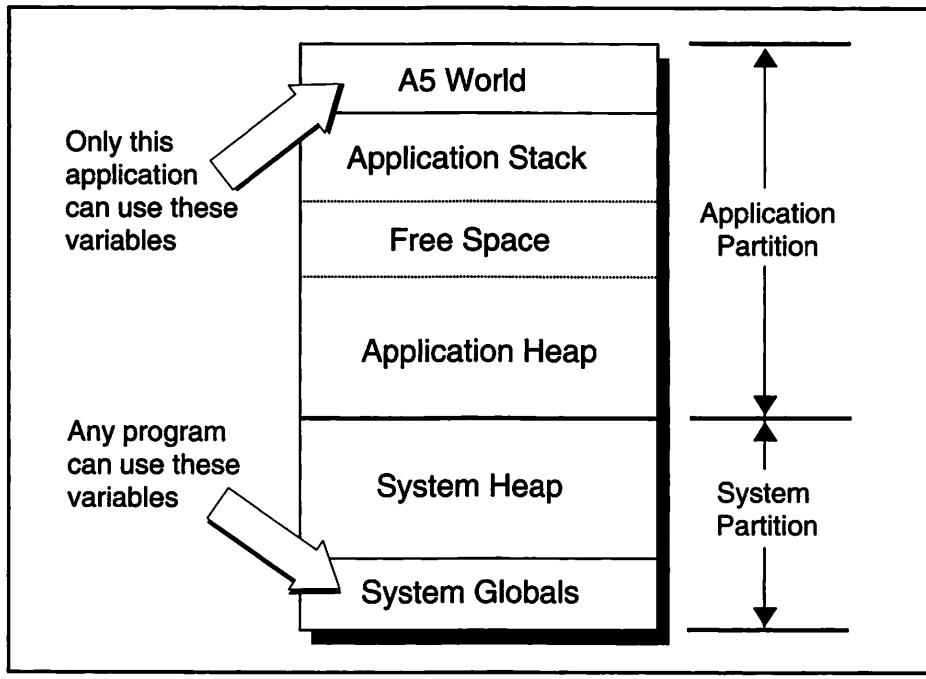


Figure 2–3. *The A5 World*

The number of global variables in any single program is fixed. Upon loading a program into the application partition, the operating system can determine the exact amount of memory it should allot to the A5 World. For this reason, the size of the A5 World is fixed when an application is loaded. The exact number of local variables and passed parameters in a program are not as well-defined. Variables local to functions are created and destroyed dynamically as the program executes. This necessitates a stack that is capable of growing and shrinking in size.

The bottom of the stack is fixed in memory, and is "anchored" just under the A5 World. As the stack adds variables, it grows downward in memory. As the stack removes variables the stack recedes back upwards. Variables are always added and removed from the top of the stack. Figure 2–4 shows the application stack. The shaded arrow emphasizes that as the stack grows it moves toward the application heap.

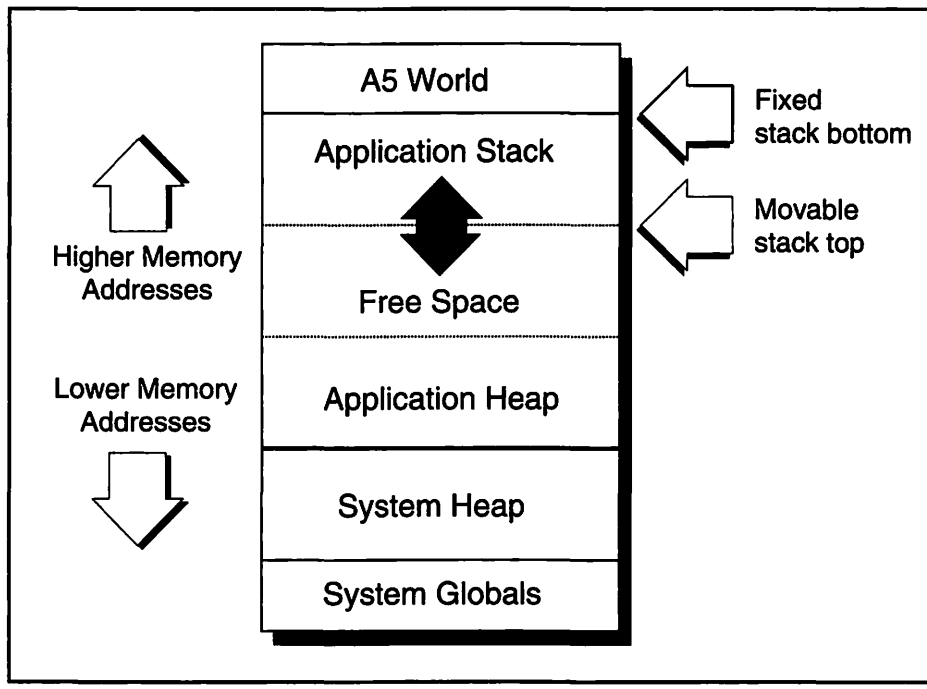


Figure 2–4. *The application stack*

Application heap

The third component of the application partition is the *application heap*. The heap holds the executable code of the application and the application's resources. Unlike the stack, which stores variables in a linear manner, the heap is capable of loading, storing, and unloading objects—program code and resources—anywhere in the area of memory that the system has established as the heap.

The application heap, like the stack, is capable of growing and shrinking as it needs more space. In this respect the application heap differs from the system heap, which takes on a fixed size when you start your computer. The application heap grows upward in memory, towards the stack. This is shown in Figure 2–5.

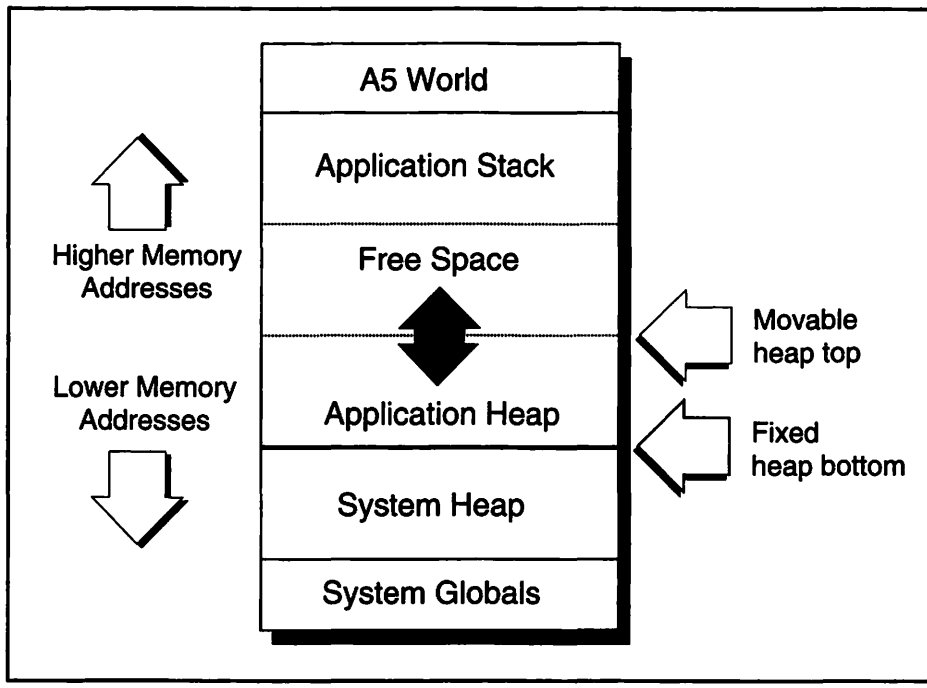


Figure 2-5. *The application heap*

The *Memory Manager*, the set of system routines that handle memory-related tasks, is responsible for allocating and managing *blocks*, or sections, of memory in the heap. These blocks hold objects such as resources. The Memory Manager can vary the *attributes*, or properties, of individual blocks of memory. Some of these attributes are: locked or unlocked, purgeable or unpurgeable, relocatable or nonrelocatable. Chapter 9 discusses these attributes in more detail.

Now that you know that the stack can grow down towards the heap, and the heap can grow up towards the stack, a question may come to mind. What prevents the stack and heap from running into one another? The answer: sometimes they do! The Memory Manager does its best to prevent this from occurring, and you can assist the manager by using some simple memory management techniques discussed in this chapter.

Summary of memory organization

Figure 2–6 summarizes several ideas and terms unique to Macintosh memory organization.

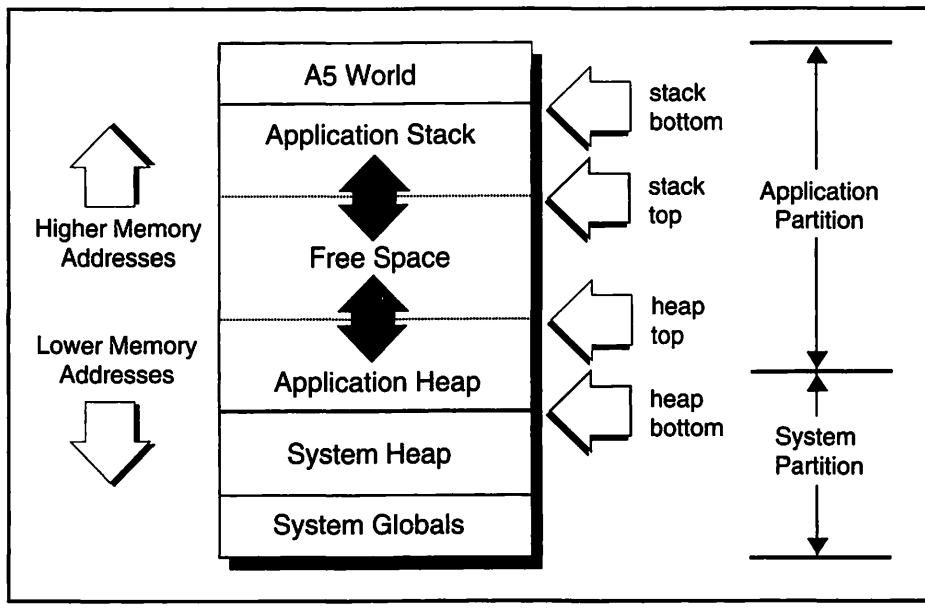


Figure 2–6. *Memory organization summary*

Up to this point, the discussions on memory have centered on examples that have just a single application running. System 7, and MultiFinder before that, allow a user to have multiple programs running at one time. Each program that runs gets its own application partition, and each partition has its own A5 World, application stack, and application heap. Figure 2–7 shows memory when two applications are running.

As a programmer, you will have no control of, nor will you be very interested in, what happens in the system partition. Any program that you create for the Macintosh will end up in an application partition when it executes. However, you will be interested in the memory management of application partitions. For this reason the topics in the remainder of this chapter apply only to application partitions. Of particular importance is the area of memory where your program's code and resources reside—the application heap.

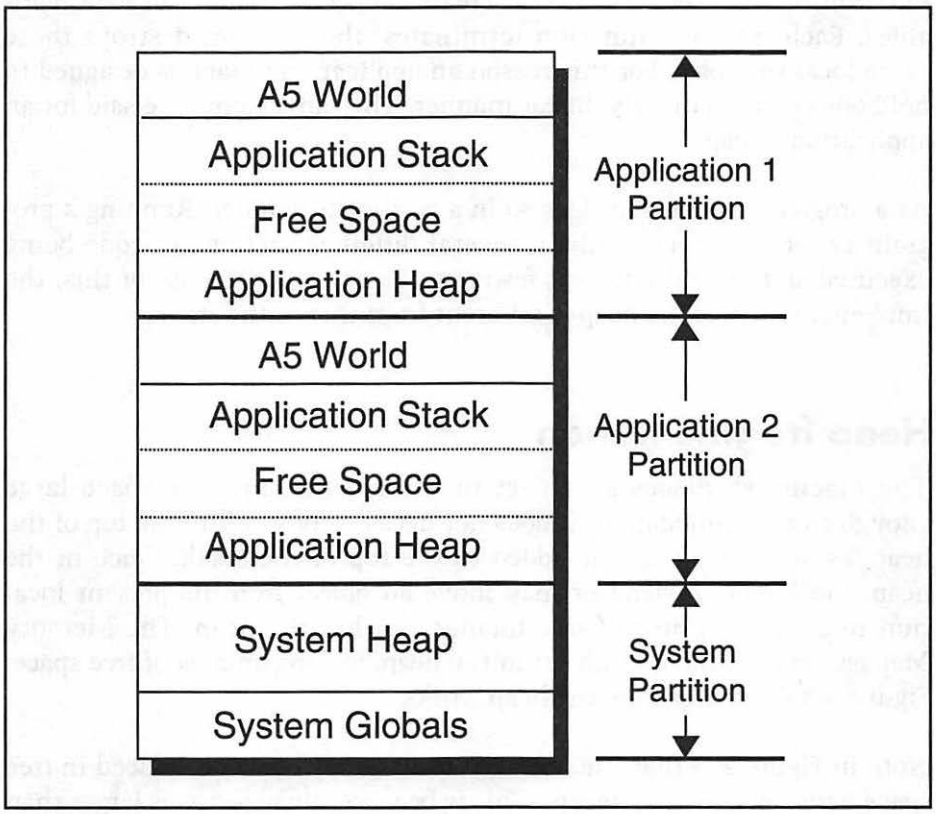


Figure 2-7. *Memory organization when more than one application is running*

Lesson 2-1: Memory Organization

You can run the program enclosed with this book for a hands-on tutorial about this topic.

The Application Heap

For a given application, certain things will remain constant each time the application is executed. When an application calls a particular function it will always pass the same number of parameters. Each time the

function begins execution it will create the same number of local variables. Each time the function terminates, the program destroys these same local variables. For this reason an application's stack is designed to hold objects in an orderly, linear manner. The same cannot be said for an application's heap.

As a program executes, it does so in a nonlinear manner. Running a program two times may result in several different sections of code being executed and several different resources being used. Because of this, the implementation of the heap is different from that of the stack.

Heap fragmentation

The Macintosh places an object in the heap in any free space large enough to accommodate it. It does not necessarily add it to the top of the heap, as an object must be added to the top of the stack. Once in the heap, the Memory Manager may move an object from its present location to a different area of free memory within the heap. The Memory Manager may remove an object in the heap, leaving an area of free space. Figure 2–8 illustrates how the heap works.

Note in Figure 2–8 that the object that was added was not placed in free space between existing objects. This is because the object was larger than either of the two free areas. When the Memory Manager adds an object to the heap it always places it in contiguous memory—it never divides it. This results in heap memory that goes unused.

When objects break up free space, *fragmentation* occurs. Several small areas of memory will be free but, due to their small individual size, they will go unused. This is shown in Figure 2–9.

Lesson 2–2: Heap Fragmentation

You can run the program enclosed with this book for a hands-on tutorial about this topic.

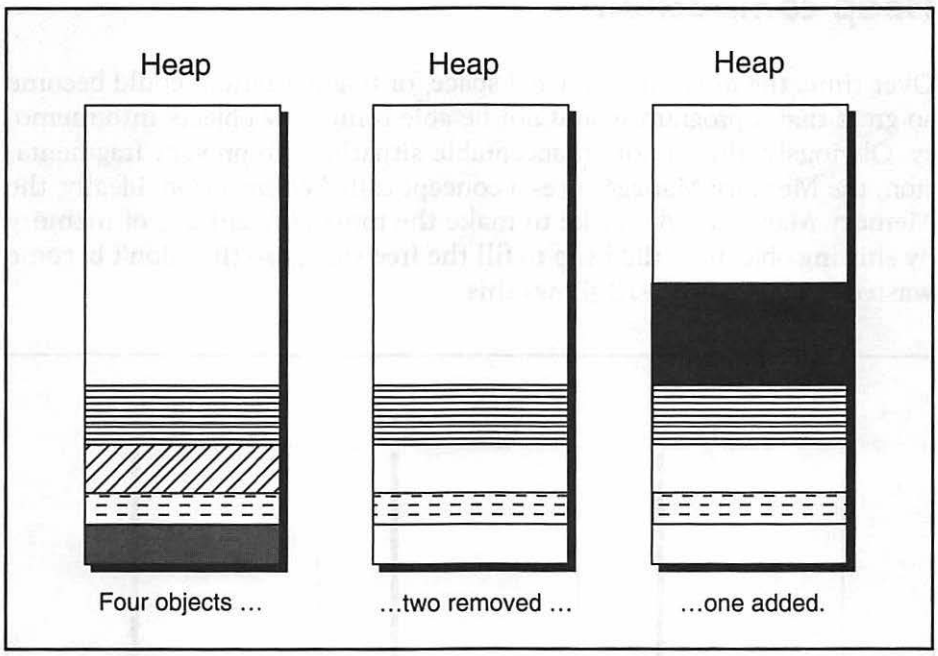


Figure 2–8. *How the heap gets fragmented*

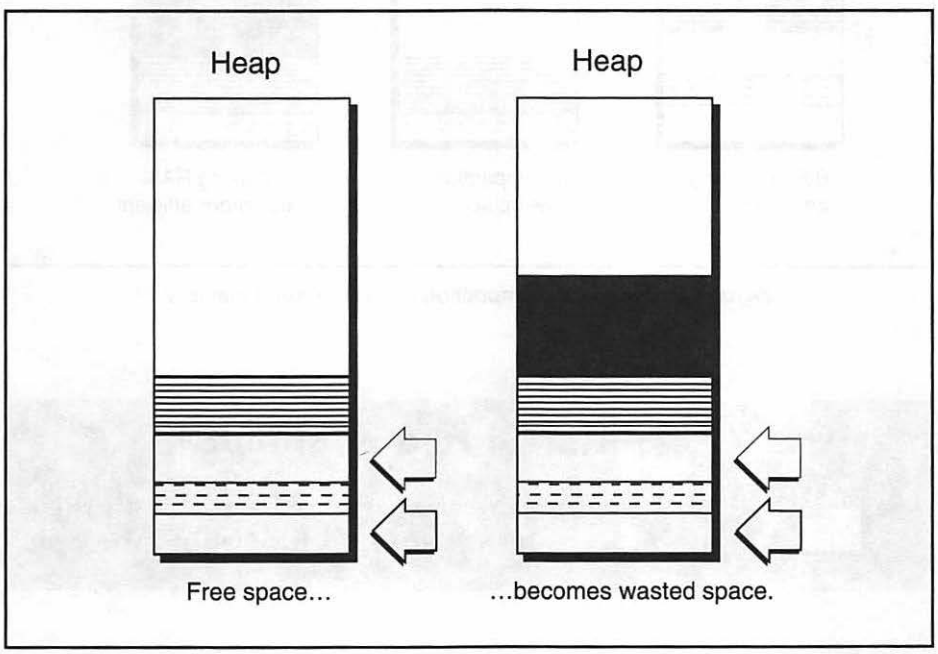


Figure 2–9. *Fragmentation creates wasted memory*

Heap compaction

Over time, the amount of wasted space, or fragmentation, could become so great that a program would not be able bring new objects into memory. Obviously, this is not an acceptable situation. To prevent fragmentation, the Memory Manager uses a concept called *compaction*. Ideally, the Memory Manager would like to make the most efficient use of memory by shifting objects in the heap to fill the free spaces so they don't become wasted RAM. Figure 2–10 shows this.

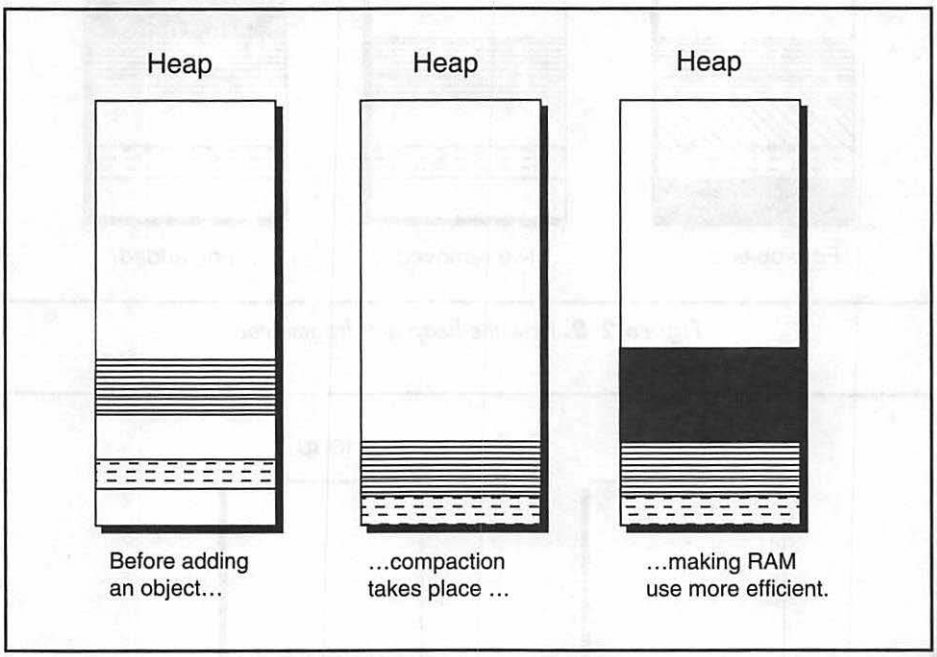


Figure 2–10. *Heap compaction restores wasted memory*

Lesson 2–3: Heap Compaction

You can run the program enclosed with this book for a hands-on tutorial about this topic.

During memory compaction the Memory Manager may decide to *purge*, or remove from memory, some blocks. Only blocks that are not currently in use, and that are specifically marked as purgeable, can be removed.

Nonrelocatable and relocatable blocks

One of the attributes of a block is whether the block is marked as relocatable or nonrelocatable. The Memory Manager can move blocks that are *relocatable* from one area of the heap to another. Blocks that are *nonrelocatable* always stay in one place, even when memory is being compacted.

Because the Memory Manager can't move nonrelocatable blocks, you might think they could cause fragmentation. And they do. Though it is vastly preferable to use relocatable blocks, there are occasions when the Macintosh must use nonrelocatable blocks. Chapter 9 covers these situations.

With all this shifting of memory taking place, how do the Memory Manager and your application keep track of where things in memory will be at any given moment? For this the Macintosh uses a technique involving master pointers. A *master pointer* is a special pointer that points to an object and stays fixed in memory, regardless of where the object to which it points moves to. If the object moves in memory, the contents of the master pointer will change to reflect the object's new address.

Figure 2–11 shows an object in memory, arbitrarily starting at memory location 65000. I labeled a few of the addresses and will reference them in the upcoming discussion. I've also labeled the application's stack, heap, and the free space that lies between these areas.

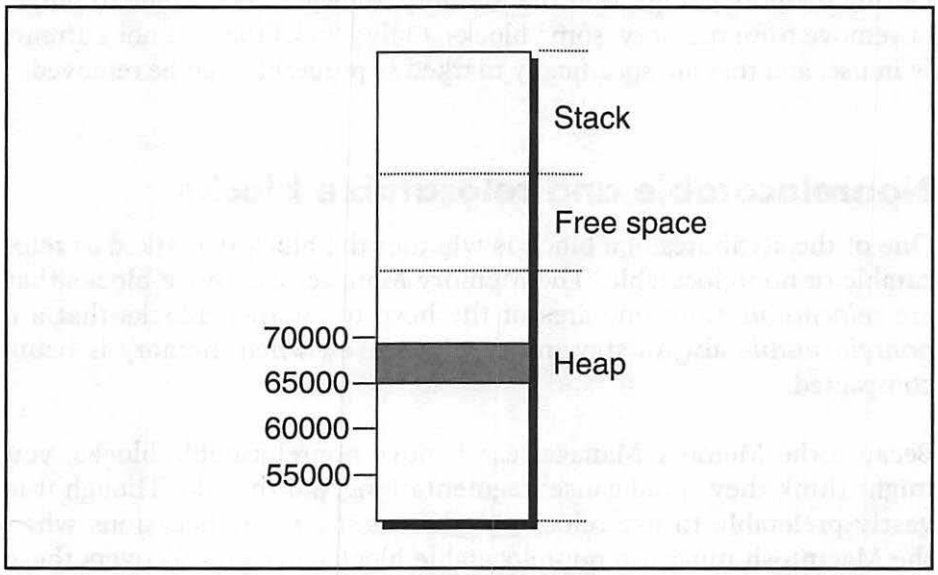


Figure 2–11. *An object in heap memory*

Figure 2–12 shows a master pointer that points to a single heap object. The master pointer contains the starting address of this object—65000.

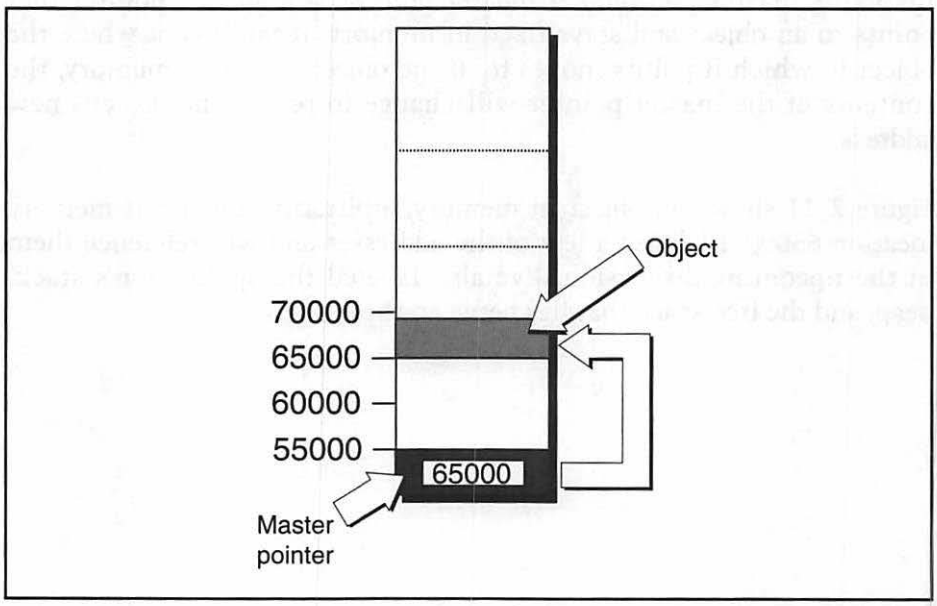


Figure 2–12. *A master pointer holds the address of an object*

The distinction between the contents of the master pointer and the address of the master pointer itself can be a source of confusion. In Figure 2–12 the content of the master pointer is 65000, while the address of the master pointer—where it is physically located in memory—is 55000.

The Memory Manager uses the master pointer to keep track of a moving object. You, the programmer, still need one other device so that the Memory Manager can relay this dynamic information to you and your program. This device is a *handle*. A handle contains the address of a master pointer. To keep tabs on a moving object in memory you will declare a handle variable in your program. Because it is a variable, it will reside on your application's stack. The handle variable will contain the address of a master pointer. Figure 2–13 illustrates this.

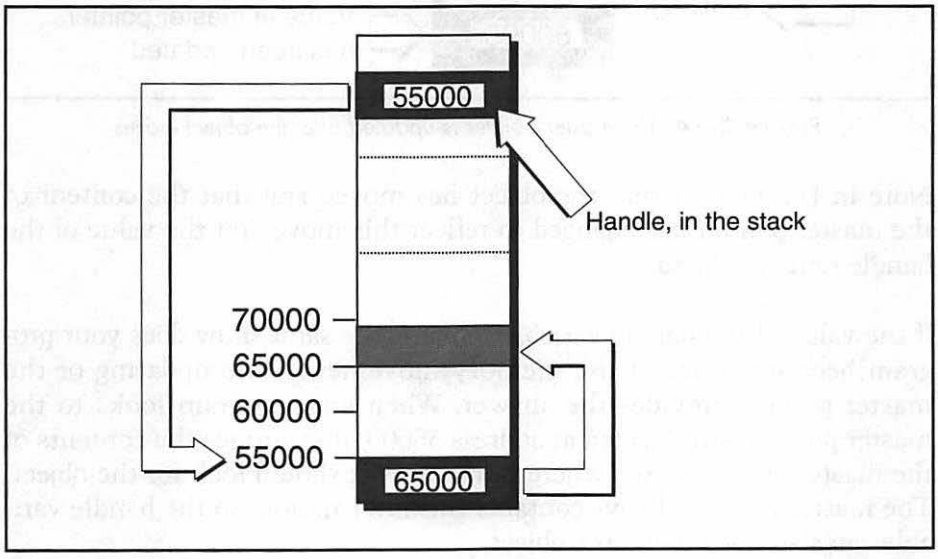


Figure 2–13. *A handle holds the address of a master pointer*

Once declared, the content, or value, of the handle variable will not change. In Figure 2–13 you can see that the handle has the value of the master pointer—55000. Because the master pointer never moves, the handle's value will never change.

If the Memory Manager compacts memory, the value held in the master pointer will change. In Figure 2–14 the object in memory is moved from address 65000 to address 60000.

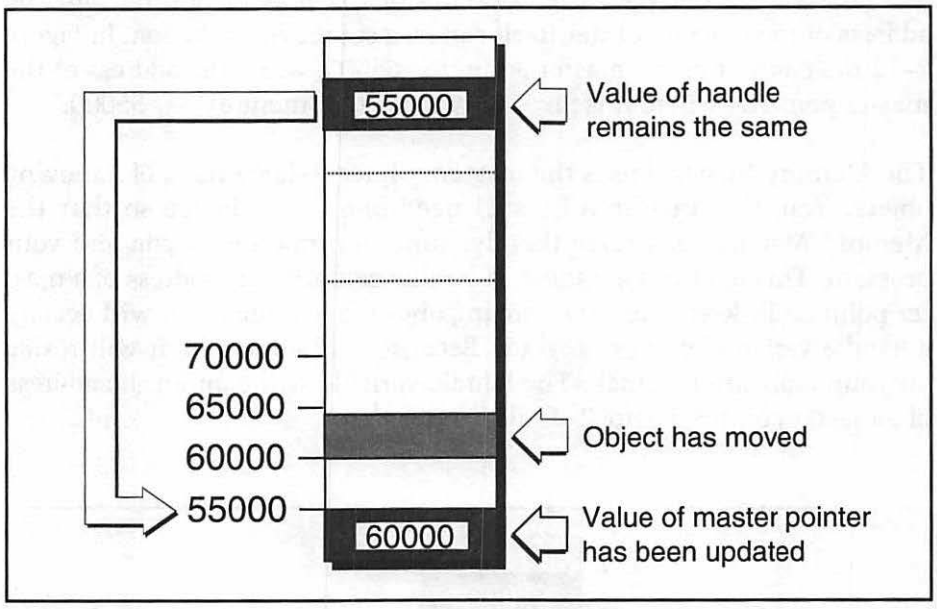


Figure 2–14. *The master pointer is updated after the object moves*

Note in Figure 2–14 that the object has moved and that the content of the master pointer has changed to reflect this move. Yet the value of the handle remains the same.

If the value of the handle variable remains the same, how does your program become aware of the memory movement? The updating of the master pointer provides the answer. When your program looks to the master pointer, still located at address 55000, it examines the contents of the master pointer to see where in memory it should look for the object. The master pointer always contains this information, so the handle variable can also track down the object.

Lesson 2–4: Master Pointers and Handlers

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Chapter Program: Memory Partitions

This chapter's example program is a simple demonstration that proves your computer really is setting aside a separate partition for each program. *MemoryFiller* is a copy of last chapter's example program *VeryBasics*—with one change. Instead of opening one window like *VeryBasics*, *MemoryFiller* uses a *for* loop to open 30 windows.

A window from *MemoryFiller* doesn't take up a lot of memory—less than 100 bytes in fact. So 30 windows should fit in less than 3K of memory. If you're like most Mac owners, your computer has at least 1 Meg of memory—1000K—and perhaps much more. So there should be no reason why running the *MemoryFiller* should result in your Mac running out of memory, right? Wrong.

Whether *MemoryFiller*, or any other program runs successfully or not isn't dependent on the total amount of memory in your computer. It depends on the amount of memory allocated to the partition that will hold *MemoryFiller.*

The THINK C compiler used to create the programs in this book normally sets the partition size of a program you create to 384K. When I used the THINK C compiler to turn the *MemoryFiller* source code into an application, I set its partition size to just 16K. If you use THINK C you can choose "Set Project Type" from the Project menu to bring up the dialog shown in Figure 2–15. Type in the desired size in Kilobytes.

If you have System 7 on your Mac, try running the *MemoryFiller* program included on the disk. If you have some version of System 6.0, first make sure your computer is running MultiFinder. From the desktop, choose "Set Startup" from the Special menu. You'll see a dialog like the one shown in Figure 2–16. If the MultiFinder radio button isn't selected click on it, then dismiss the dialog and reboot your computer.

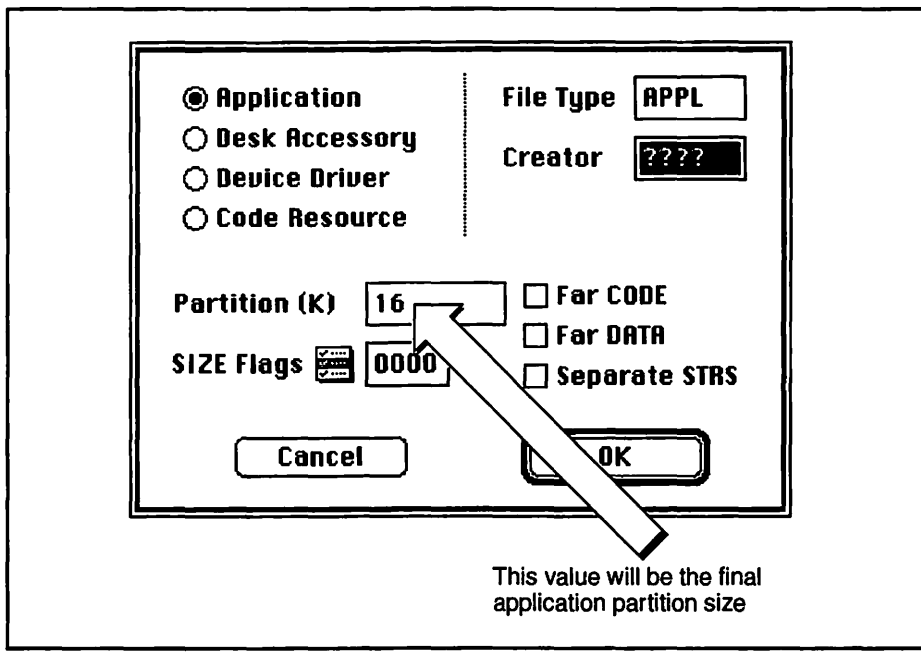


Figure 2-15. *Establishing a program's partition size in THINK C*

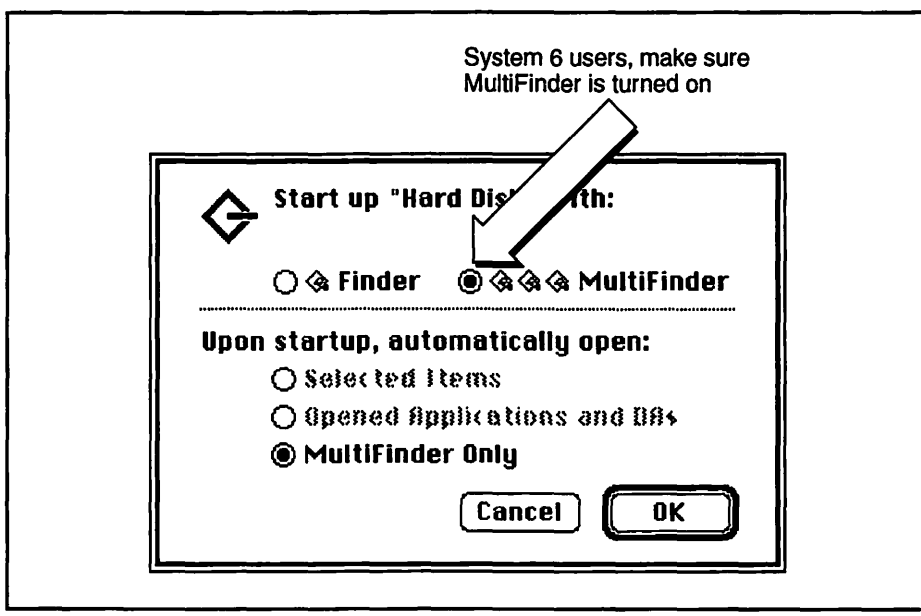


Figure 2-16. *For System 6, make sure MultiFinder is running*

When *MemoryFiller* starts, the program will begin to quickly put windows on the screen, one on top of another. You can tell by what appears to be a flashing title bar on the window. Each flash is actually a new window being placed over the previous one. You'll probably count about 8 or 10 windows before the program suddenly quits and returns to the Finder. There it will display a dialog like one of those in Figure 2–17. I looked up "error type 25" in a list of system errors and found that it is an "out of memory" error, as you probably suspected.

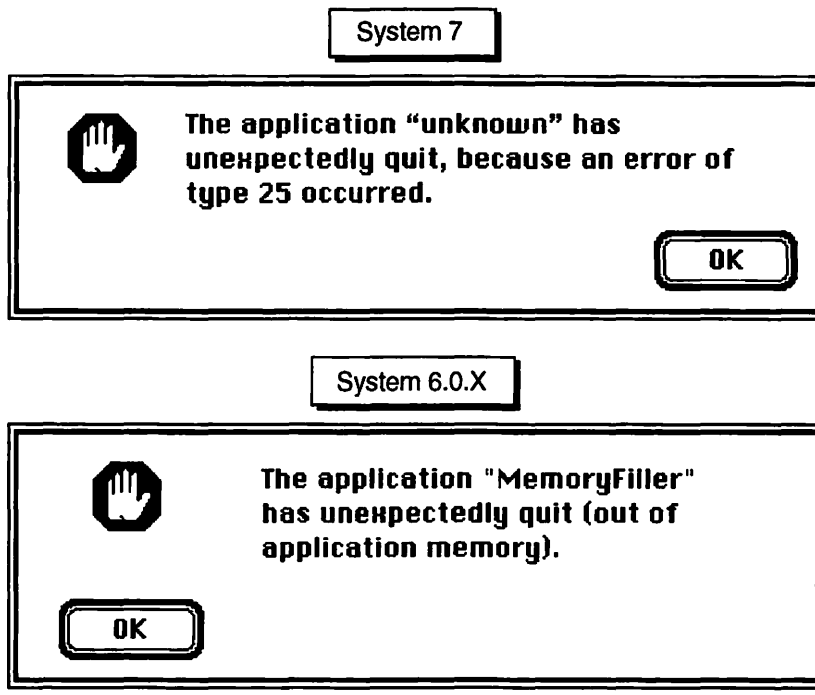


Figure 2–17. *"Out of Memory" error messages*

You can get the MemoryFiller program to run by clicking once on its icon and then selecting "Get Info" from the File menu or typing Command-I. The dialog that appears lets you change the application's partition size. Try setting the partition to 32K. If you're using System 6, type in 32 in the Application Memory Size edit box. System 7 users should type 32 in the Preferred size edit box. Figure 2–18 shows both situations.

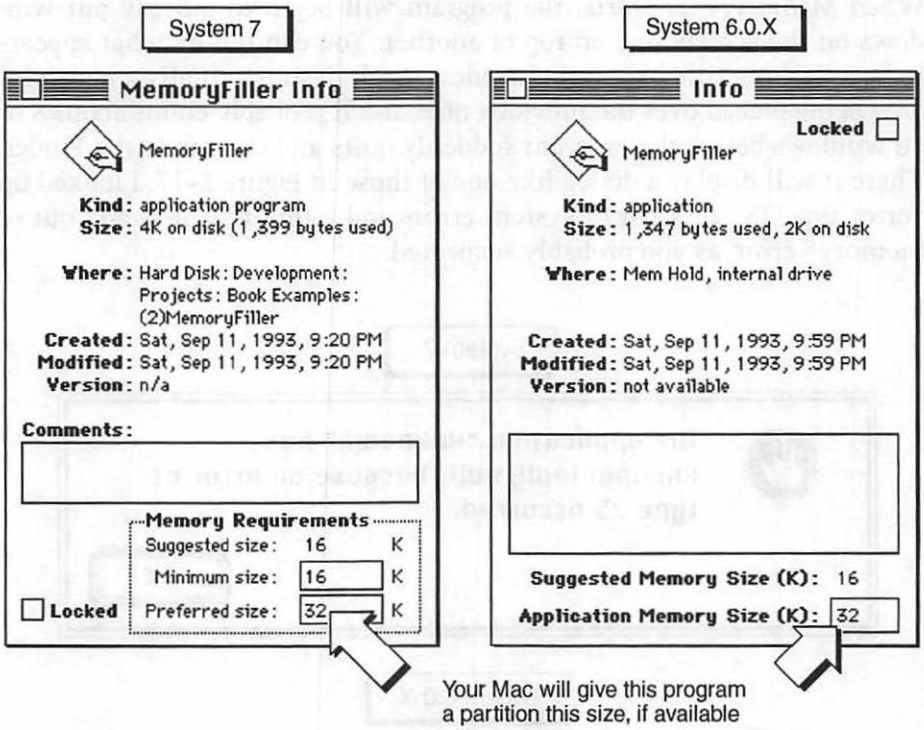


Figure 2-18. *"Get Info" for both System 7 and System 6.0.X*

With the partition size changed, rerun *MemoryFiller*. This time all 30 windows open. Click the mouse once to end the program.

With one megabyte or more of memory at your disposal, you saw a program that needs just a little over 16 K refuse to run. This should show you that memory partitions are indeed very real.

Program resources: *MemoryFiller.π.rsrc*

The resource file for *MemoryFiller* is identical to that of last chapter's *VeryBasics* resource file. In fact, I simply copied the *VeryBasics* file and renamed it! It has just one 'WIND' resource. Remember, resources act as templates. Even though thirty windows will be opened, only one 'WIND' resource is needed. If you want the windows to be of a different style or size from one another, you would need more 'WIND' resources.

Program listing: MemoryFiller.c

As mentioned, the source code for *MemoryFiller* is almost identical to that of *VeryBasics*.

```
/*++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void  Handle_Mouse_Down( void );


/*+++++++++++++   Define global constants   +++++++++++++++++++*/

#define    WIND_ID                    128
#define    BEEP_DURATION                1
#define    NIL                         0L
#define    IN_FRONT       (WindowPtr)-1L
#define    REMOVE_EVENTS                0


/*+++++++++++++++   Define global variables   +++++++++++++++++*/

WindowPtr    The_Window;
Boolean      All_Done = FALSE;
EventRecord  The_Event;
short        i;


/*++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();

   for ( i = 0; i < 30; i++ )
      The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

   SetPort( The_Window );
```

```
    MoveTo( 30,50 );
    DrawString( "\pChapter One Program" );

    while ( All_Done == FALSE )
    {
       GetNextEvent( everyEvent, &The_Event );

       switch ( The_Event.what )
       {
          case mouseDown:
             Handle_Mouse_Down();
             break;
       }
    }
}


/*++++++++++  Handle a click of the mouse button  ++++++++++*/

void  Handle_Mouse_Down( void )
{
   SysBeep( 5 );
   All_Done = TRUE;
}
```

Stepping through the code

MemoryFiller.c places the call to *GetNewWindow()* in a *for* loop so that it gets called 30 times:

```
for ( i = 0; i < 30; i++ )
   The_Window = GetNewWindow(WIND_ID, NIL, IN_FRONT);
```

All of the remaining code is identical to the code of *VeryBasics*. If you have any questions about any of the lines, refer back to Chapter 1.

Chapter Summary

The Macintosh Operating System divides RAM into two main sections, or partitions. It reserves one partition, the system partition, for its own use. The other partition is dedicated to applications that you run. This second partition is further subdivided into application partitions. There is one application partition for every application that's running.

An application partition is composed of three main areas: the A5 World, the application stack, and the application heap. The A5 World is used to store a program's global variables. The application stack is used to hold a program's local variables. Finally, the application heap is used to hold the bulk of a program: its resources, including the program's code resources.

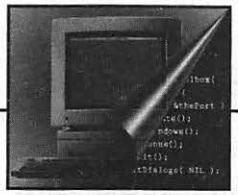
The Memory Manager is the set of system routines that allocate the blocks, or sections, of memory. A block of memory is capable of holding many different things, such as program code or other resources. This book generically refers to these "things" as objects.

Each block of memory has attributes, or characteristics, that can be set. Whether a block is relocatable, or movable in memory, is one such attribute. Other attributes are discussed in Chapter 9.

The section of memory called the application heap is the area of most interest to a Macintosh programmer. Because some memory blocks can be moved about in the heap, the heap can become fragmented—areas of memory develop that are too small to fit an object. One technique that the Memory Manager performs periodically on its own is compaction; that is, blocks are rearranged to eliminate small pockets of wasted space that lie between them. The next chapter covers programming techniques you can use to help the Memory Manager minimize fragmentation.

Because blocks of memory that are purgeable can be shifted about in memory, a special memory management technique is necessary to keep track of blocks. A master pointer is a special pointer that holds the address of a single object. Though the object it points to can be moved, the master pointer itself never moves. Instead, when the block the master pointer points to is moved, the contents of the master pointer are simply updated to reflect the object's new location.

The programs you write will have to keep track of where objects reside in memory. A handle is a variable that allows a program to keep track of an object that moves about in memory. Once declared, the value of a handle doesn't change. That's because a handle holds the address of a master pointer, which itself is a nonmoving object.



3 Resources

In Chapter 1 I implied that almost everything a Macintosh does involves resources. In fact, this book could consist of one huge chapter called "Macintosh Resources." A more manageable approach, and the one that this book takes, is to mention each resource in the chapter to which it best pertains. Additionally, one chapter is devoted to resources that aren't clearly dependent on one subject.

The 'WIND' resource type, mentioned in chapters 1 and 2, defines a window. Chapter 5 covers this resource in more detail. Dialog boxes use the 'DITL' and 'DLOG' resource types, and the 'ALRT' resource defines an alert. Chapter 6 covers these three resource types. Chapter 7 discusses the 'MENU' and 'MBAR' resource types that create menus.

This chapter deals with storing strings of text in a 'STR#'—a resource that is a list of strings. You load them into memory and display them in a window.

Here you will see an easy way to include pictures in your programs through the 'PICT' resource. In addition, you will learn how you can use a series of pictures to create animation.

The Macintosh has built-in hardware that makes it ideal for playing sound. The 'snd ' resource is a way for you to store a prerecorded sound along with your program.

Every program has its own icon that the Finder displays on the desktop. You can give your application its own distinctive icon by creating a 'BNDL' resource.

About Resources

Before getting down to actual examples, I present a short summary of the importance of resources and their proper use in your applications.

The importance of resources

Chapter 1 provided a good summary of exactly what resources are. I'd like to reiterate that almost everything either starts out as a resource or ends up as a resource.

You use a resource editor like Apple's ResEdit or Mathemaesthetics' Resourcerer to define the features of your program's windows, dialogs, menus, and alerts in a resource file. When your compiler turns your source code into an executable program it converts the source code into 'CODE' resources. It then packages the 'CODE' resources together with the resources in your resource file to get the final standalone Macintosh program.

Resources provide Macintosh programs with a uniform look. Every program for the Mac has menus and windows that look similar. This makes users new to an application comfortable. Having resources that are editable with a graphical editor such as ResEdit makes it easy to change

Resource types

There are about 100 different resource types. You'll probably only need to use fewer than a dozen types in your programs. Here's a list of some of the more common ones.

'ALRT'	Defines the look of an alert box
'BNDL'	Relates an icon to a program
'CODE'	All the instructions of a program
'DITL'	Contents of a dialog box
'DLOG'	Defines the look of a dialog box
'DRVR'	Desk accessory—a driver
'ICN#'	List of icons
'PICT'	Picture
'SIZE'	Partition size of a program
'STR#'	List of strings
'WIND'	Defines the look of a window
'snd '	Sound

Checking for errors

One big advantage of resources is that they are easy to edit. Anyone can easily modify and remove a program's resources. However, the fact that they are easy to edit is also a disadvantage. When your program wants to makes use of a resource it expects to be able to find a resource of the proper type and ID. If that resource doesn't exist, the Toolbox call that is looking for it will fail.

A failed Toolbox call can spell disaster for your program. The application may quit and return to the user to the desktop, or, worse yet, it may freeze the Macintosh if the programmer has not implemented proper error-handling.

Failure to put up any of the key elements of a program, such as the menu bar, a window, or a dialog, will spell disaster. Subsequent calls to other routines depend on these elements being in place. You'll always want to verify that the calls that load these pieces of your program are successful.

Toolbox calls that involve resources often return a pointer or a handle to the program. The *GetNewWindow()* routine is one example. It returns a *WindowPtr*. Up to this point I haven't put effort into verifying that a Toolbox call did indeed return a valid pointer. Here's an example of what I've been doing:

```
#define    WIND_ID                 128
#define    NIL                      0L
#define    IN_FRONT    (WindowPtr)-1L

WindowPtr  The_Window;

The_Window = GetNewWindow(WIND_ID, NIL, IN_FRONT);

[ go on our merry way... ]
```

Now that you know better, I'll use a little error checking from here on. Here's how I'll handle the above *GetNewWindow()* call.

```
The_Window = GetNewWindow(WIND_ID, NIL, IN_FRONT);
if ( The_Window == NIL )
   ExitToShell();

[ now go on our merry way...]
```

If there is insufficient memory to load the window, or the specified 'WIND' resource doesn't exist, the call to *GetNewWindow()* will fail and the *WindowPtr* will be nil. The *if* statement checks for this condition. Should the pointer be nil, the code will exit the program and return to the desktop through the use of the Toolbox routine *ExitToShell()*. You

now have assurance that a call to *GetNewWindow()* that fails will not result in a frozen screen.

This error-handling technique is minimal at best. What you should really do is write an error handling routine that puts up an alert with a descriptive message in it. That gives the user some information about the problem and a chance to correct the ugly situation. At the very least, it enables the user to describe the problem in a much more detailed manner to someone else.

The example program that appears in the final chapter of this book is the most comprehensive example I give. That program incorporates full-fledged error checking.

Working With Strings

One of the major advantages of a program that relies heavily on resources, as all Macintosh programs do, is that you can make many changes to a program even after compiling it. Apple recommends that programmers store all displayable text as resources. Then, if you want to make a version of your program usable by non-English speaking people; for example, you can edit the text within the program's resources. Depending on other factors in your program, you might not even have to change any source code or recompile your program. This, of course, is easier said than done; in practice, *internationalizing*, or localizing, an application is a more involved matter. But resource editing provides a very good start.

The 'STR#' resource

The 'STR#' resource is a list of strings, each one up to 255 characters in length. Figure 3–1 shows how ResEdit displays the two strings in the 'STR#' resource from this chapter's example program. Although there are other third-party resource editors available, the examples in this book will use Apple's ResEdit. ResEdit is widely available at no cost to programmers, and it does the job.

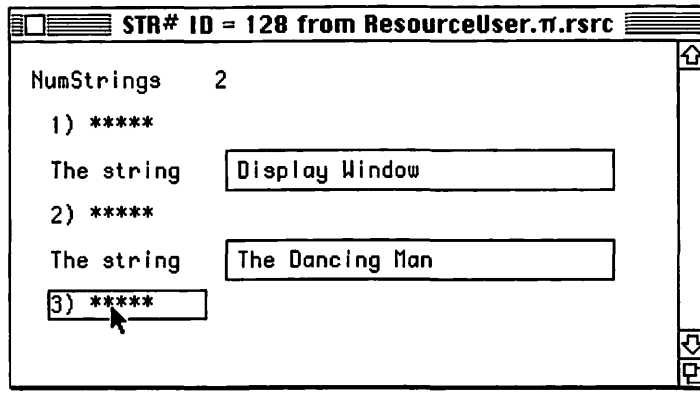


Figure 3–1. *A 'STR#' resource with two strings*

To create the 'STR#' resource, select "Create New Resource" from the Resource menu in ResEdit. You'll then see the "Select New Type" dialog box. Scroll to the 'STR#' type and double click on it.

To add a new string to the 'STR#', first click the mouse button on the number that appears in the window. Figure 3–1 adds a third string by doing just that. Next, select "Insert New Field(s)" from the Resource menu. ResEdit will respond by adding an edit box in which you can type your new string.

Using a string in a program

To use a string stored as a resource, your code must first load it in to memory with a call to the Toolbox routine *GetIndString()*. Because a 'STR#' contains more than one string, you must specify which string in the list you want. This is done by including an index to the string. The first string in the 'STR#' is 1, the second is 2, and so forth.

The call to *GetIndString()* sets a variable of *Str255* type equal to the resource string. The *Str255* is a Macintosh type that holds a string of up to 255 characters. Here's a call to *GetIndString()* that sets *the_str* equal to whatever string is listed second in a 'STR#' with ID of 128:

```
Str255  the_str;

GetIndString( the_str, 128, 2 );
```

Typically you'll add a *#define* for the resource ID of the 'STR#' list and a descriptive *#define* for each string in the string list. The 'STR#' in Figure 3–1 has two strings in it. We will use the first as the title of a window and the second string later as a label for a picture we'll put in the window.

Here's an example that opens a window and writes a string to the window by setting the variable *the_str* to the second string in a string list and then writing that string to the location specified by *MoveTo()*.

```
#define   STR_LIST_ID        128
#define   WIND_TITLE_STR       1
#define   PICT_LABEL_STR       2
#define   PICT_LABEL_STR_L    70
#define   PICT_LABEL_STR_B   140

Str255     the_str;
WindowPtr  The_Window;

The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
if ( The_Window == NIL )
   ExitToShell();

SetPort( The_Window );

GetIndString( the_str, STR_LIST_ID, PICT_LABEL_STR );

MoveTo( PICT_LABEL_STR_L, PICT_LABEL_STR_B );
DrawString( the_str );
```

Get in the habit of always making a call to *SetPort()* before drawing to a window. The next chapter explains ports in more detail.

Here's a second example. In this code fragment you open a window and get the first string from a string list. Instead of writing the string in the window, you use it to change the title of the window. Every window has a title in the drag bar. If you don't specify the title it will default to

"Untitled". As shown here, you can use the Toolbox routine *SetWTitle()* to change the title.

```
#define   STR_LIST_ID          128
#define   WIND_TITLE_STR         1
#define   PICT_LABEL_STR         2

Str255     the_str;
WindowPtr  The_Window;

The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
if ( The_Window == NIL )
   ExitToShell();

GetIndString( the_str, STR_LIST_ID, WIND_TITLE_STR );

SetWTitle( The_Window, the_str );
```

Figure 3–2 shows the window resulting from this code fragment.

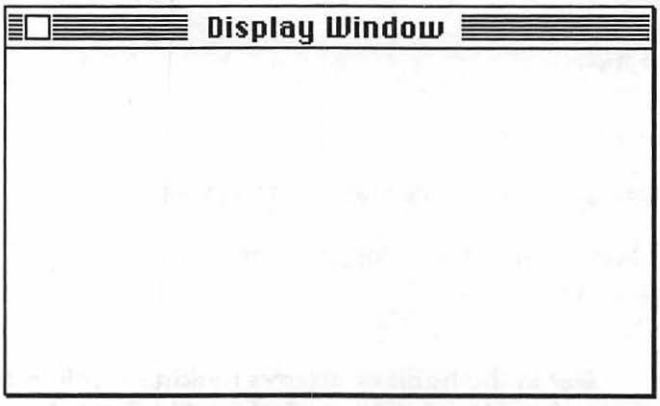


Figure 3–2. *Changing a window's title*

Lesson 3–1: The 'STR#' Resource

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Pictures and Animation

The 'PICT' resource is the Macintosh way of storing graphical images for use by a program. A program can display pictures in its windows and dialogs. You can also use pictures to easily add simple animation to your programs.

The 'PICT' resource

If you have a drawing or painting application, you can create a 'PICT' resource. MacPaint, MacDraw, Canvas, and PixelPaint are just a few examples of programs you can use. After you draw a picture, or find a piece of clip art you like, just select it from within your paint program and copy it to the Scrapbook. Save as many pictures to the Scrapbook as you want. Then run ResEdit. Once you're in ResEdit copy one of the pictures from the Scrapbook and paste it into your resource file. ResEdit will save it as a 'PICT'.

Figure 3–3 shows a simple picture in a drawing program. If you follow the above procedure for transferring the picture to ResEdit, your resource file will have a new resource type in it—a 'PICT', as shown in Figure 3–4.

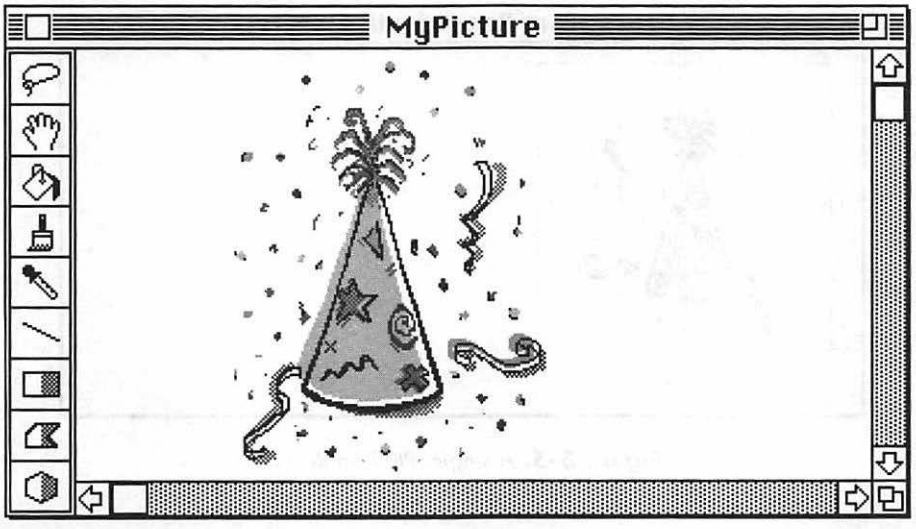


Figure 3–3. *A picture in a Macintosh paint program*

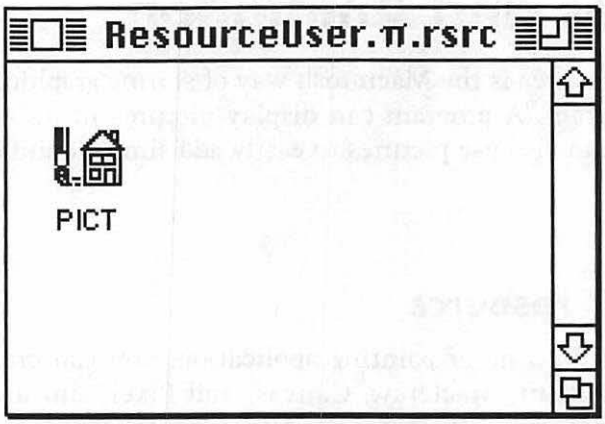


Figure 3–4. *'PICT' in ResEdit*

Double-clicking on the 'PICT' icon of Figure 3–4 will open a window that displays all of the 'PICT's in the resource file. Our example has just one, as shown in Figure 3–5. No matter how big the picture that you paste is, ResEdit will display it in a small rectangle like that of Figure 3–5. ResEdit will scale the picture as best it can. This shrunken version is for display only. If you double-click on 'PICT' 128 you'll see it at its actual size.

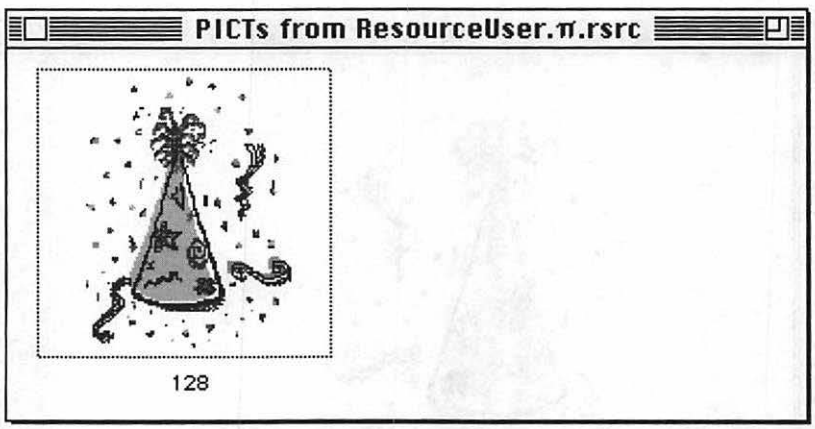


Figure 3–5. *A single 'PICT' in ResEdit*

Displaying a 'PICT' in a program

Now that you have a picture safely tucked away into a resource file, you can display it in a program.

You know all about handles from Chapter 2. Programs on the Macintosh have a special handle for working with pictures—the *PicHandle*. To load a 'PICT' resource into memory, you make a call to the Toolbox routine *GetPicture()*. This routine returns a *PicHandle* for use by your program. Here's an example:

```
#define     PARTY_HAT_PICT     128

PicHandle   hat_pict_handle;

hat_pict_handle = GetPicture( PARTY_HAT_PICT );
```

GetPicture() brings a 'PICT' into memory; it doesn't display the picture. To do that you make a call to *DrawPicture()*, which requires two parameters: a handle to a picture, and a rectangle in which to display the picture.

You can display a picture in a rectangle of any size. *DrawPicture()* will attempt to scale the original picture to fit the rectangle. But if you want to display the picture in its original, actual size, you'll need to determine that size.

A *PicHandle* is a handle to a structure called a *Picture*. One of the members of this structure is the *picFrame*, which is a *Rect* that surrounds the picture. It holds the size of the picture. To access the *picFrame*, you dereference the *PicHandle*. Let's add to the previous code fragment to see how this is done.

```
#define     PARTY_HAT_PICT     128

PicHandle   hat_pict_handle;
Rect        pict_rect;

hat_pict_handle = GetPicture( PARTY_HAT_PICT );

pict_rect = ( **( hat_pict_handle ) ).picFrame;
```

Now you have the rectangle that bounds the original picture. Your real interest is in the picture's size. You want to set up a rectangle of the proper size to display the picture anywhere in a dialog or window.

Recall from Chapter 1 that a *Rect* is a structure with four members—right, left, top, and bottom. Use the Toolbox routine *SetRect()* to set the pixel coordinates of a rectangle. Pass *SetRect()* a pointer to a *Rect* variable along with the pixel boundaries you want the rectangle to have. The order of the boundaries is important. Here's an example that sets the upper left corner of a rectangle at coordinates (75, 40) and a width of 100 and a height of 50.

```
#define   LEFT    75
#define   TOP     40
#define   RIGHT   175
#define   BOTTOM  90

SetRect( &the_rect, LEFT, TOP, RIGHT, BOTTOM );
```

The upper left corner of the window is the reference point for the rectangle's boundaries. Figure 3–6 shows where the rectangle would be located for the above example. The figure uses a dashed line to show the rectangle because *SetRect()* only sets up a rectangle—it doesn't actually display one.

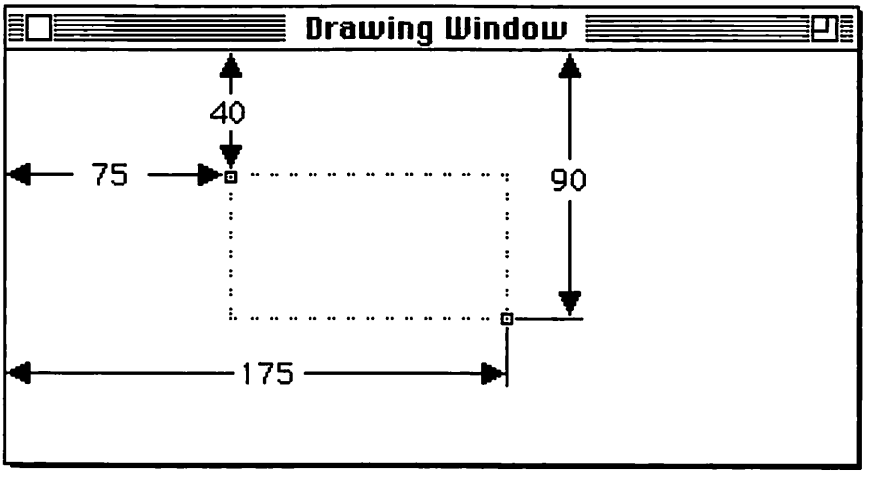


Figure 3–6. SetRect() *boundaries are relative to a window*

Now you know how to get the original rectangle that holds the picture boundaries and how to set up your own rectangle using *SetRect()*. Use the values of the original bounding rectangle to determine the picture's height and width. Use those values also to set up a rectangle the size of the picture anywhere you want. Finally, display the picture in your rectangle using *DrawPicture()*. Here's a complete example.

```
#define     PARTY_HAT_PICT     128

WindowPtr   The_Window;
PicHandle   hat_pict_handle;
Rect        pict_rect;
short       pict_wd;
short       pict_ht;

The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
if ( The_Window == NIL )
   ExitToShell();

SetPort( The_Window );

hat_pict_handle = GetPicture(PARTY_HAT_PICT);

pict_rect = (**(hat_pict_handle)).picFrame;

pict_wd  = pict_rect.right - pict_rect.left;
pict_ht  = pict_rect.bottom - pict_rect.top;

SetRect(&pict_rect, 100, 50, 100 + pict_wd, 50 + pict_ht);
DrawPicture(hat_pict_handle, &pict_rect);
```

This example loads the resource 'PICT' with an ID of 128 into memory using *GetPicture()*. It then dereferences the *PicHandle* that *GetPicture()* returned in order to access the *picFrame*. The width and height of the original picture are determined from the picFrame. A rectangle is then set up to display the picture. This rectangle starts 100 pixels in from the left of a window and 50 pixels down from the top. The width and height of the rectangle are the same as those of the original 'PICT'. Finally, the picture is displayed in the window with a call to *DrawPicture()*. Figure 3–7 shows this.

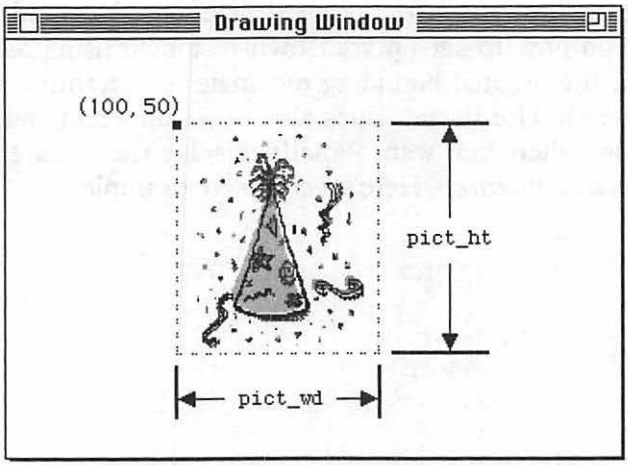


Figure 3–7. *Placing a picture in a window*

Now you know exactly how to create a picture, save it as a 'PICT' resource, and display it in the window of a program. With just a little more work you can use 'PICT's to really add a little excitement to your applications, especially in the form of animation.

Lesson 3–1: The 'PICT' Resource

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Using 'PICT's to create animation

You can create animated effects in your programs by displaying a series of 'PICT's, one after another, from within a loop in your source code. To do this you first create a series of 'PICT's, then write a routine that brings these 'PICT's into memory and displays them in a window.

Creating a series of 'PICT's

Figure 3–8 shows a screen shot of a document from a popular Macintosh paint program. In the figure, I showed off my drawing expertise by drawing four characters, each in a different pose. I actually only drew the left-

most character. I then copied him and used the paint programs free rotate feature to shift the character to a slightly different pose.

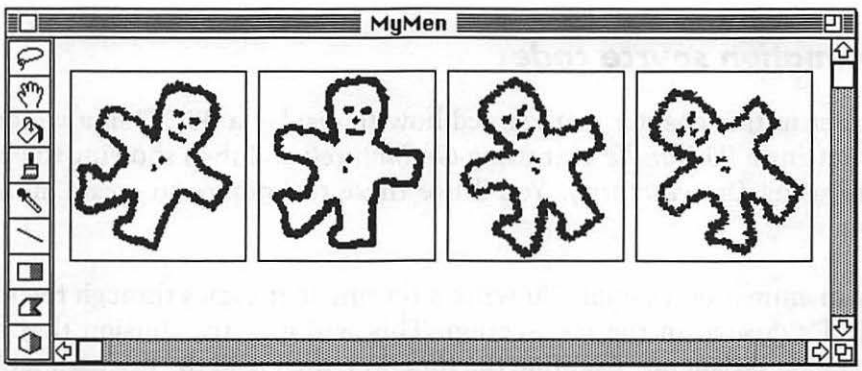


Figure 3–8. *Scenes for animation drawn in a paint program*

The characters in Figure 3–8 have a frame surrounding them for one reason only—so that each will be the same size when you copy them individually to the Scrapbook. When copying a single character I made the selection just within, and not including, the border. After copying all four pictures to the Scrapbook I ran ResEdit and copied each picture out of the Scrapbook and into a resource file. When you double-click on the file, the 'PICT' resource opens to a window that displays the four 'PICT's, as shown in Figure 3–9.

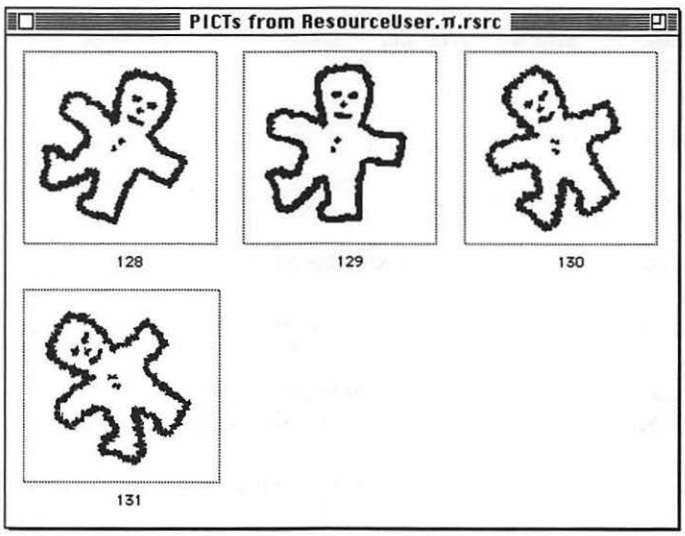


Figure 3–9. *ResEdit after pasting four 'PICT's into a resource file*

After taking note of the resource IDs of the 'PICT's, quit ResEdit and run the compiler. It's time to write some code.

Animation source code

Earlier in this chapter you learned how to display a 'PICT' in a window by getting a *PicHandle* to it using *GetPicture()* and then showing the picture using *DrawPicture()*. You'll use these techniques to create animation.

As an animation example I'll write a routine that cycles through the four 'PICT's created in the last section. This will give the illusion that the character is dancing. Examine the function, then read the discussion of it that follows.

```
#define  FIRST_MAN_PICT   128
#define  PICT_L            30
#define  PICT_T            20
#define  DELAY_TICKS        7

void  Draw_Moving_Picture( void )
{
   Rect        pict_rect;
   PicHandle   pict_handle;
   short       pict_wd, pict_ht;
   short       i, count;
   short       pict_id;
   long        end_tick;

   SetPort( The_Window );

   pict_handle = GetPicture( FIRST_MAN_PICT );

   pict_rect = ( **( pict_handle ) ).picFrame;

   pict_wd  = pict_rect.right  - pict_rect.left;
   pict_ht  = pict_rect.bottom - pict_rect.top;

   SetRect( &pict_rect, PICT_L, PICT_T, PICT_L+pict_wd, PICT_T+pict_ht );

   count = 0;
```

```
    for (i=1; i < 31; i++)
    {
       ++count;
       switch ( count )
       {
          case 1:
             pict_id = FIRST_MAN_PICT;
             break;
          case 2:
             pict_id = FIRST_MAN_PICT + 1;
             break;
          case 6:
             pict_id = FIRST_MAN_PICT + 1;
             count = 0;
             break;
          case 3:
          case 5:
             pict_id = FIRST_MAN_PICT + 2;
             break;
          case 4:
             pict_id = FIRST_MAN_PICT + 3;
             break;
       }

       pict_handle = GetPicture(pict_id);
       DrawPicture(pict_handle, &pict_rect);

       Delay(DELAY_TICKS, &end_tick);
    }
}
```

Much of *Draw_Moving_Picture()* should look familiar to you. It uses *GetPicture()* to get a handle to one of the 'PICT's for the purpose of determining its size. You'll use this size to display each of the four 'PICT's—thus the importance of making them all the same size in your drawing program.

The heart of *Draw_Moving_Picture()* is the *for* loop. The loop executes 30 times. You can choose to make it execute as few or as many times as you want. Within the loop, I use the variable *count* to keep track of which of the four pictures is to be displayed. If I just continually cycled through the four 'PICT's in order, the animation would look jerky after showing the fourth picture and then jumping back to the first. So, I use a different approach. After displaying the fourth 'PICT' I backtrack, dis-

playing the third, then second, and finally the first picture. It's similar to the motion of a pendulum. Figure 3–10 elaborates on this plan.

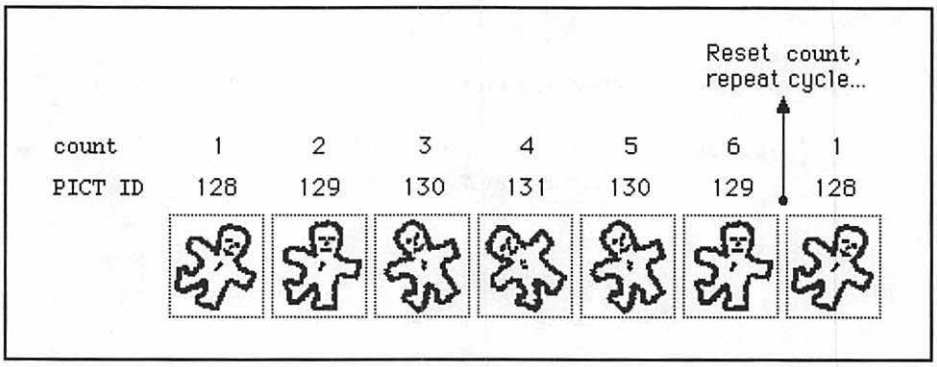


Figure 3–10. *Animation: cycling through the 'PICT's*

After using a variable count to determine which 'PICT' to use, I get a handle to the correct 'PICT' and then display it—right on top of the previous 'PICT'. That way I don't have to bother erasing the previous picture.

After drawing one picture, and before displaying the next, I've included a delay in the program. Some Macintosh computers can run through this loop very quickly—our little man would be really dancing up a storm. I use the Toolbox function *Delay()* to cause a short delay between pictures to slow things down.

The *Delay()* function requires two parameters. The first is the length of the delay. Give this in sixtieths of a second increments—that's how the Macintosh keeps track of time. Thus a value of 1 results in a delay of one-sixtieth of a second, while a value of 120 results in a two-second delay. You'll usually ignore the second parameter to *Delay()*. This is a pointer to a variable of type *long*. When the *Delay()* routine has finished, the Toolbox will have filled this variable with the time, in sixtieths of a second, since the system was started; that is, since the Macintosh was turned on.

Lesson 3-3: Animation

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Sounding Off

The Macintosh has built in sound hardware and software that allows it to easily play quality sound. Chapter 1 mentioned that the Macintosh has several managers–groups of related Toolbox routines that do much of the behind the scenes work for you. The Sound Manager is one such manager. Routines in the Sound Manager allow you to play 'snd ' resources.

All resource types are composed of four characters. If it appears that a type is only three characters, like the 'snd ' type, then it ends with a space. The space is required.

The 'snd ' file

A 'snd ' resource is your means of transferring a recorded sound to other Mac users and your means of including sound in your programs. A 'snd ' resource can exist as a separate file—that's how you can give or receive one from other people. It can also exist embedded in a program's resource file. That way it can be transferred with that program and used by that program.

You can record your own 'snd ' resources or obtain them from others. On-line bulletin boards like CompuServe, GEnie, and America Online have hundreds, perhaps thousands, of 'snd ' resources. They were recorded and uploaded by other Mac users so that all Mac users would have a source of free sounds to download and play on their Macs.

Be aware that a downloaded 'snd ' is usually in a space-saving stuffed, or compacted, format. Programs such as StuffIt Expander exist to unstuff these compacted files. If you don't already have StuffIt Expander you can download it from these bulletin boards. Figure 3–11 shows a folder that contains StuffIt Expander, a stuffed 'snd ' resource, and the same resource after unstuffing with StuffIt Expander. I downloaded both StuffIt Expander and the Laughing Elvis.sit 'snd ' from America Online.

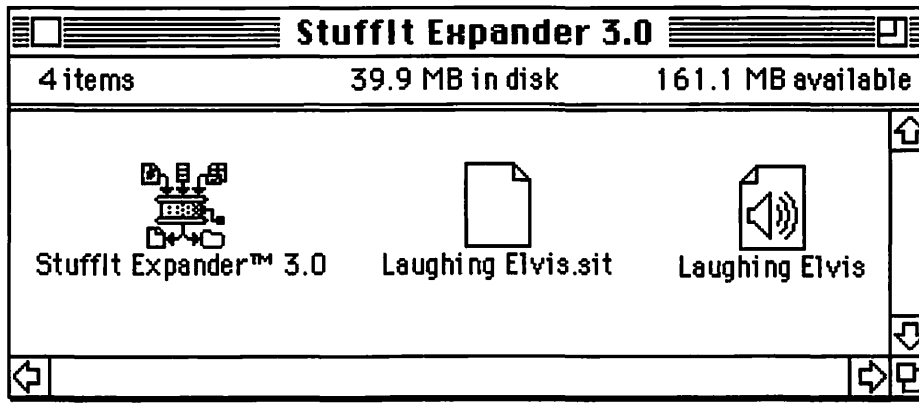


Figure 3–11. *StuffIt Expander, a stuffed 'snd ' , and a 'snd ' resource*

From Figure 3–11 you can see that the Laughing Elvis 'snd ' looks much like any other Macintosh file. That is what makes a 'snd ' so useful. You can copy Laughing Elvis to a floppy disk and give it to anyone else who has a Macintosh.

The 'snd ' resource

Once you have a file that contains a 'snd ' you'll want to get the sound into the resource file of your program. Then you'll be able to write source code that loads the sound into memory and plays it. As usual, ResEdit is the tool for the job.

To copy a sound from a file to a resource file, first run ResEdit. From the File menu select "Open" and open the 'snd ' file (make sure to open the unstuffed file). It will contain one 'snd ' resource. Click on it and then copy it. Next, open the resource file of your program. Paste in the copied sound. That's all there is to it. Figure 3–12 shows the three sounds included in this chapter's example program.

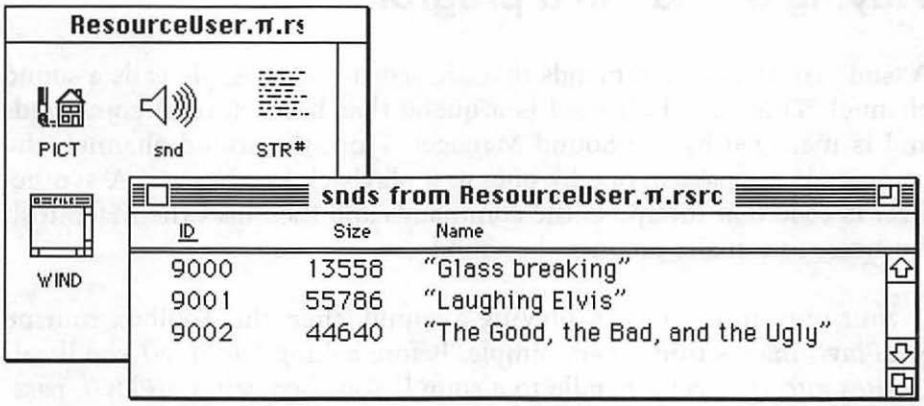


Figure 3–12. *'snd ' resources*

A 'snd ' , like any resource, has an ID. Apple has reserved ID numbers 0 to 8191 for its own use. You can change the ID of a 'snd ' by selecting "Get Info" from ResEdit's File menu. You'll then see a dialog like that in Figure 3–13. There you can type in a new ID. Taking into consideration the reserved numbers, I chose to number the three sounds starting at ID 9000.

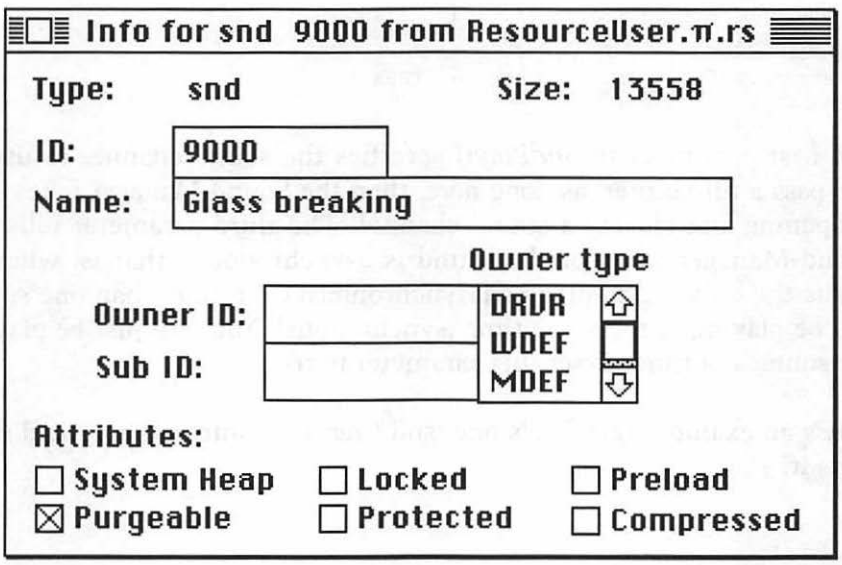


Figure 3–13. *The Get Info window for a 'snd' resource*

Playing a 'snd ' in a program

A 'snd ' consists of commands that are sent to what Apple calls a sound channel. This sound channel is a queue that holds several commands and is managed by the Sound Manager. From the sound channel, the commands are passed, one by one, to a playback synthesizer. A synthesizer is code that interprets the commands and then uses the Macintosh hardware to actually produce the sound.

If your only interest is in playing a sound, then the Toolbox routine *SndPlay()* makes things very simple. Before calling *SndPlay()* you'll call *GetResource()* to get a handle to a sound. You then call *SndPlay()*, passing it the handle to the 'snd ' resource. Here's an example. Note that you need to include the *Sound.h* header file when you use *SndPlay()*.

```
#include <Sound.h>

#define   SND_GLASS_ID   9000

Handle  snd_handle;
OSErr   err;

snd_handle = GetResource( 'snd ' , SND_GLASS_ID );

err = SndPlay( NIL, snd_handle, TRUE );
```

The first parameter to *SndPlay()* specifies the sound channel to use. If you pass a nil pointer, as done here, then the Sound Manager takes care of opening and closing a sound channel. The third parameter tells the Sound Manager whether the sound is asynchronous—that is, whether this is the only sound playing (asynchronous) or if more than one sound will be playing at the same time (synchronous). You will just be playing one sound at a time, so set this parameter to true.

Here's an example that loads one 'snd ' resource into memory and then plays it.

```
#include <Sound.h>

#define   NIL              0L
#define   SND_GLASS_ID   9000
```

```
Handle  snd_handle;
OSErr   err;

snd_handle = GetResource( 'snd ' , SND_GLASS_ID );

if ( snd_handle == NIL )
   ExitToShell();

err = SndPlay( NIL, snd_handle, TRUE );

if ( err != noErr )
   ExitToShell();
```

Notice that I perform a check to see if *GetResource()* successfully retrieved the 'snd ' resource before calling *SndPlay()*. Sounds can take up a lot of memory—perhaps more memory than was allocated for your program. If this is indeed the case, the returned handle will be nil and you can exit the program.

When *SndPlay()* is finished it will return an error code to the *OSErr* variable *err*. The *OSErr* type is a Macintosh C type used for error checking. If everything goes well *SndPlay()* will return a value of 0. If *SndPlay()* fails to play the sound properly it will return a result code other than 0. This code gives some information on the type of error that occurred. Your compiler defines *noErr* to be 0, so you can check the error code after *SndPlay()* is finished and compare its value to *noErr*.

The Macintosh has the capabilities of creating, editing, and playing complex sounds. The Sound Manager contains close to 50 Toolbox routines. If your interest is merely in having your application play back prerecorded sounds, you'll only need to familiarize yourself with *SndPlay()*.

Giving a Program an Icon

When you use a Macintosh compiler to build, or create, your application it ends up with the generic icon displayed in Figure 3–14. If you want your application to display its own custom icon on the desktop, you'll need to create a 'BNDL' resource in the program's resource file before building the program.

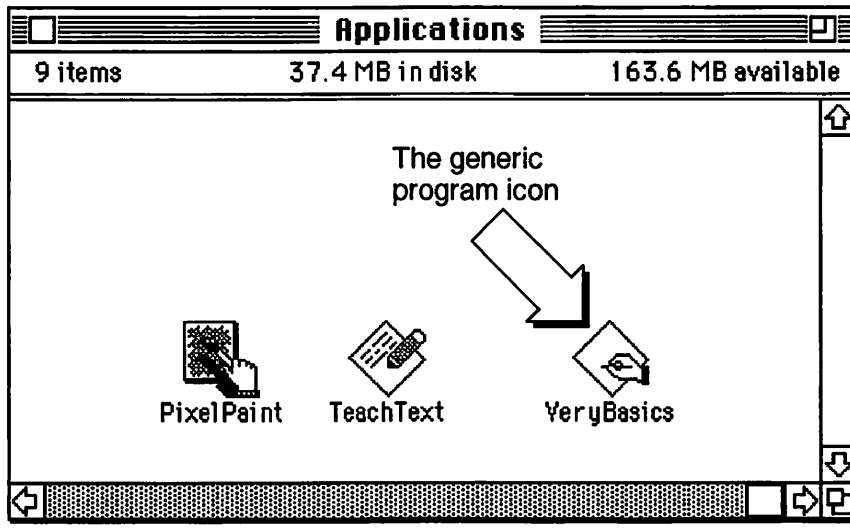


Figure 3-14. *Typical program icons on the desktop*

The Finder is responsible for displaying an icon for each program and program file that appears on the desktop. To keep track of what file gets what icon, the Finder makes use of a file's Type and Creator identifiers. The following paragraphs cover the essential background information you'll need before working with a 'BNDL' resource.

The Finder and icons

Every program has an icon, a Type identifier, and a Creator identifier. The Finder looks to a program's identifiers to see what icon it should display on the desktop to represent that program. All applications have a Type of APPL. Each application should have a four letter Creator code that is unique to that application.

You can give your application any combination of four upper- and lowercase letters for it to use as its Creator code. At the end of this chapter is a sample program called *ResourceUser*. I chose a Creator name of "Rusr", though I could have chosen any one of countless combinations of letters.

You will specify the Creator name at two times: when you create a 'BNDL' resource for the program and when you build your program.

Creating the 'BNDL' resource

In ResEdit, selecting "Create New Resource" from the Resource menu allows you to create a 'BNDL' resource. You'll be presented with the Select New Type dialog box. There, scroll to the 'BNDL' type and double click on it. When you do, you'll see a dialog like that of Figure 3–15.

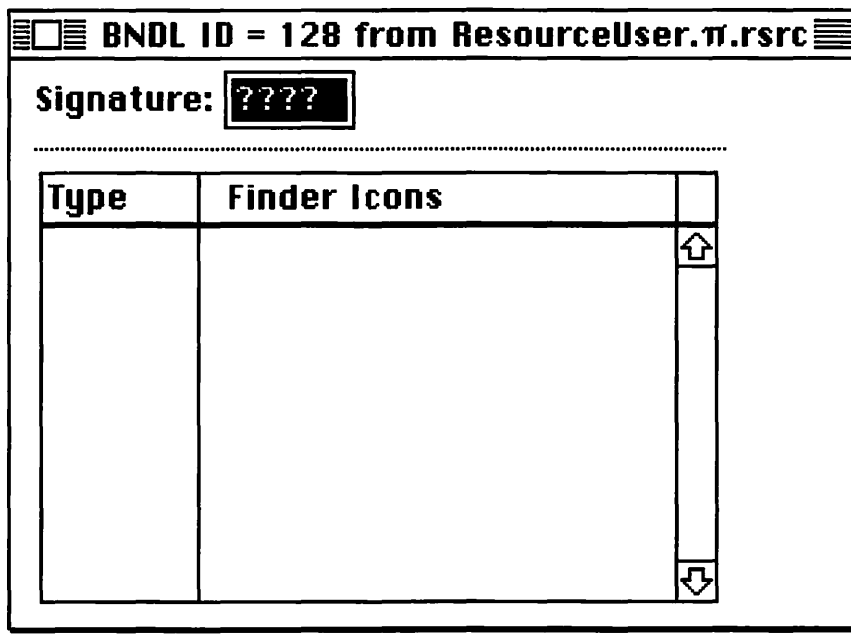


Figure 3–15. *The 'BNDL' resource in ResEdit*

In this dialog you'll enter your program's Signature, another name for the Creator. For this chapter's *ResourceUser* program you'll set the signature, or Creator, to Rusr. You want to add an icon that the Finder will display for the ResourceUser application, so select "Create New File Type" from the Resource menu. Then click the mouse under the Type column and type in APPL. The dialog will then look like that of Figure 3–16.

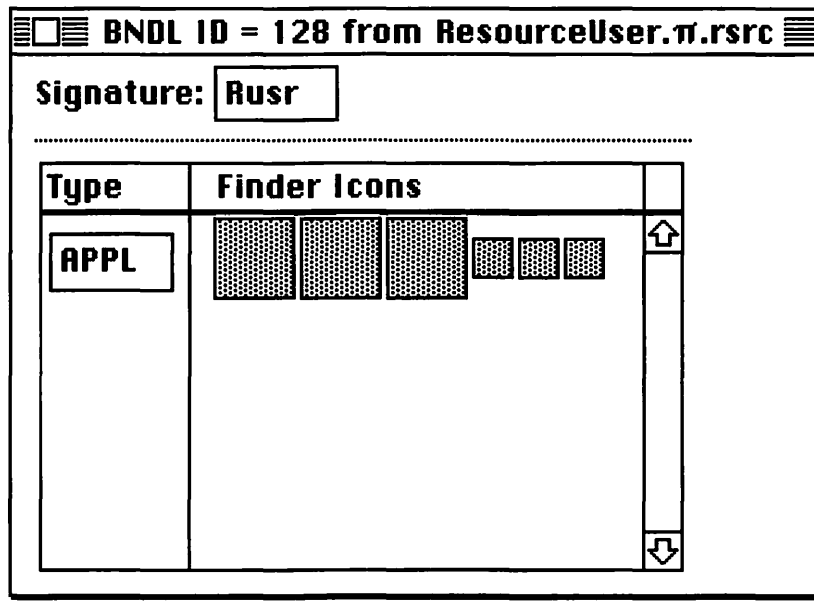


Figure 3–16. *Adding a signature and type to the 'BNDL'*

Now, let's create the icon itself. Double-click on any of the six gray boxes in the Finder Icons column. You'll see a dialog like that in Figure 3–17. You're creating the icon from scratch, so click the New button.

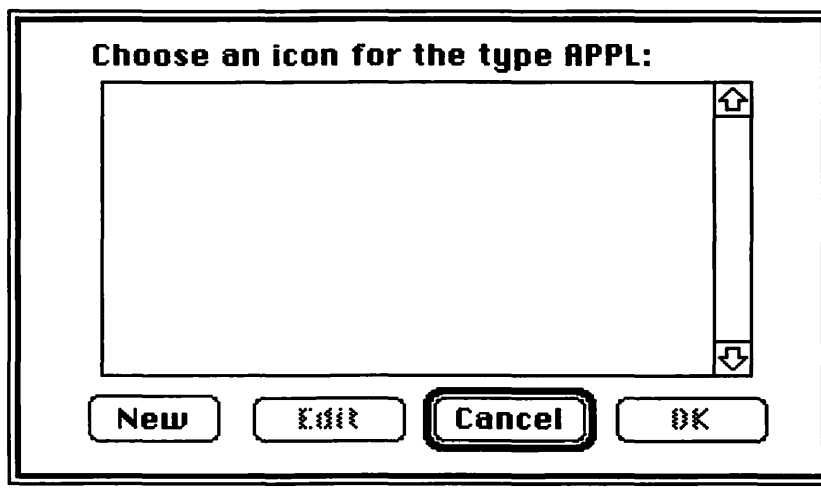


Figure 3–17. *Getting to the 'BNDL' editor in ResEdit*

Now you'll be in the 'BNDL' editor window. Here you can select a tool, such as the pencil, from the tool palette and then draw your own icon. Our ResourceUser program demonstrates how to bring resources into a program. I chose to symbolize this by drawing the ResEdit jack-in-the-box and an arrow to show the resources being brought up into the program. This is shown in Figure 3–18.

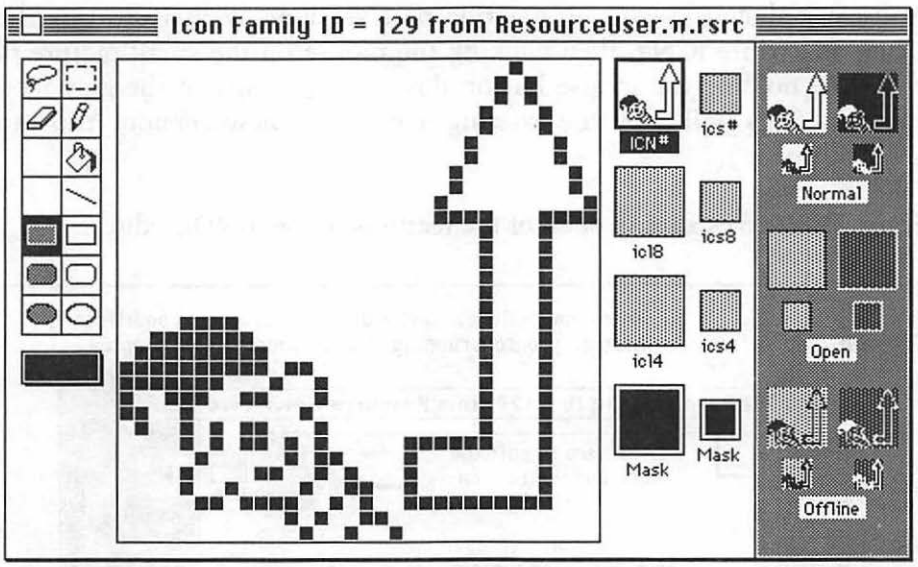


Figure 3–18. *The 'BNDL' resource in ResEdit*

As you draw the icon you'll see it displayed in actual size in the right side of the 'BNDL' editor. You may be wondering why there are so many blank icons shown there. If you want to fully accommodate users of System 7 you can create several versions of each icon. Here's a summary of what the different versions are for:

ICN#	the original icon resource that has been used for years and years. The Finder will use this version to display a black and white icon.
icl4	the Finder will diplay the icon that is here if the user has a 4-bit, or 16-color system.
icl8	if the user has an 8-bit, or 256-color system, the user will see this icon.

ics#, ics4, ics8 — the Finder sometimes displays a small icon for a program. When it does, it chooses one of these versions.

The minimum requirement for a custom program icon is that you create the ICN# version. Then, no matter what color level the user's system has, the Finder will display this black and white icon.

You can reduce the work in creating new versions by first creating the black and white ICN#, then clicking the mouse on the small picture of it. While holding the mouse button down, drag to any of the gray icon pictures. This will copy the existing icon to the new version. You can then edit it.

Figure 3–19 gives a recap of all of the features of the 'BNDL' editor.

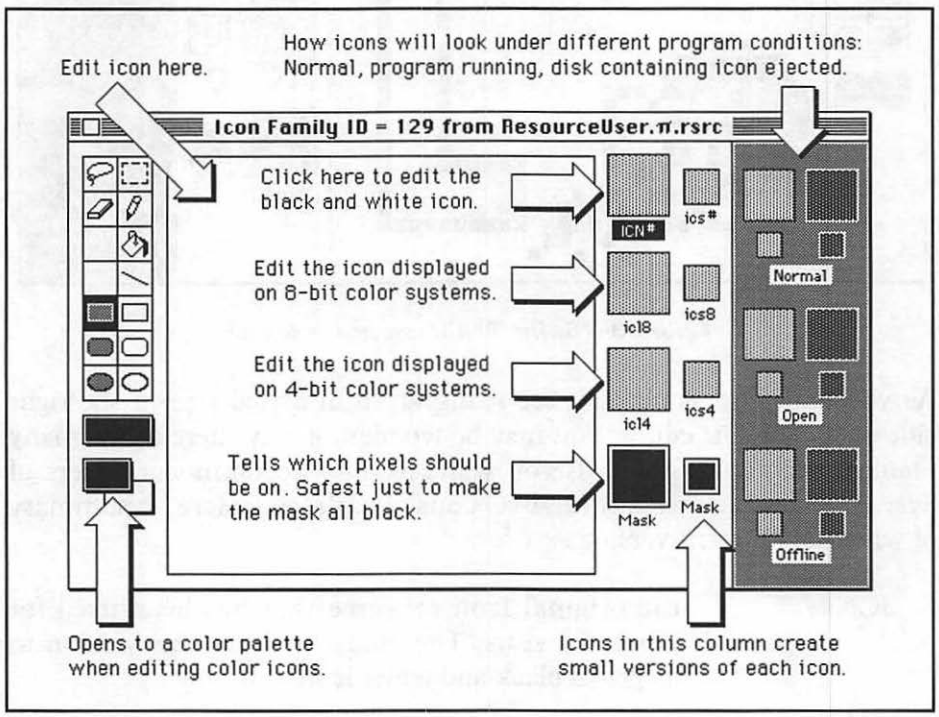


Figure 3–19. *'BNDL' editor summary of features*

When you're done with the edit, click the 'BNDL' editor's close box to return to the previous dialog. There you can view the icons you've created, as shown in Figure 3–20.

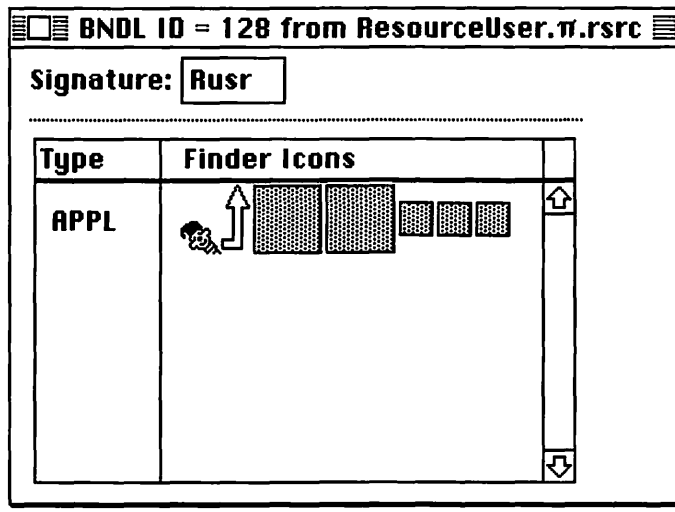


Figure 3–20. *Viewing the 'BNDL' icons*

When you look at the main window of the *ResourceUser* resource file you'll notice that there are several new resource types, as shown in Figure 3–21. Creating a 'BNDL' resource will add 'FREF' and 'ICN#' resources. It will also add a resource with the Signature, or Creator, name. If you create other icons, such as an icl8 or icl4, ResEdit will also add those resources. The 'BNDL' serves to bundle these other resources together.

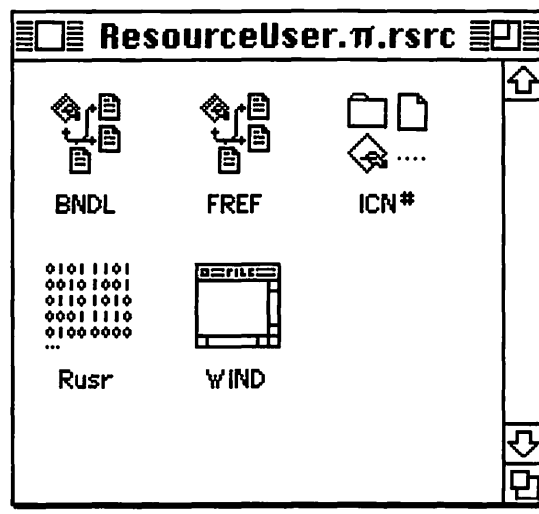


Figure 3–21. *The 'BNDL' resource in ResEdit*

Setting the Creator in the compiler

Once your 'BNDL' resource is complete you'll want to let your source code in on things. You do this by telling your compiler the Signature, or Creator, you used in the 'BNDL'.

When you build your program, your compiler lets you specify a Creator. In the THINK C environment you do this by selecting the "Set Project Type" from the Project menu. That results in the display of the dialog box shown in Figure 3–22. There you enter a four-letter Creator name.

After dismissing the dialog, you then select "Build Application..." from the Project menu to build your application.

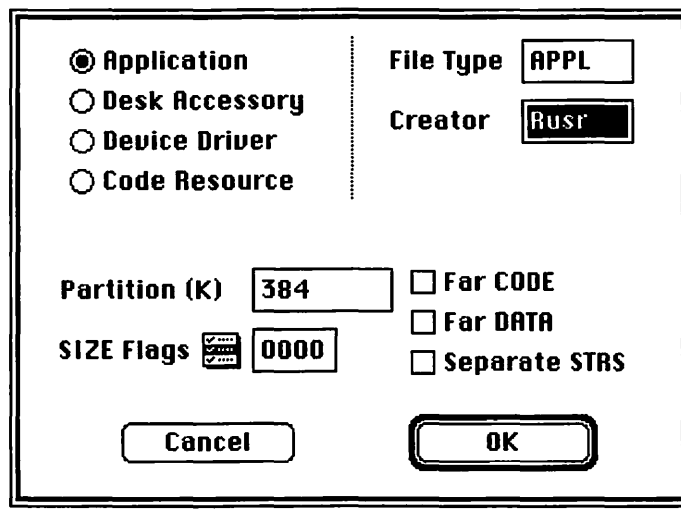


Figure 3–22. *The THINK C Set Project Type dialog*

Making the Finder aware of a new icon

After your program's resource file has a 'BNDL' resource and you've successfully done a build to create an application, there is one more step you need to take to see your application's custom icon on the desktop: you must rebuild the desktop.

The Finder stores icons in a file called the Desktop file. To get the Finder to notice a new icon you must rebuild the desktop. This scary-sounding

practice is really quite simple—you simply restart the Macintosh, holding down the Command and Option keys as the Mac starts up. You'll see a dialog like that shown in Figure 3–23. Press the OK button to continue.

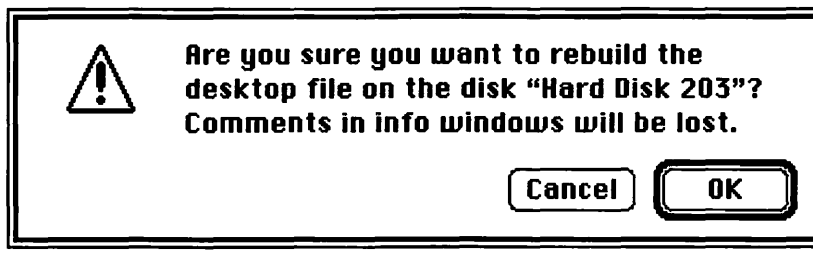


Figure 3–23. *A last chance to back out of the desktop rebuild*

From the dialog in Figure 3–23, you can see that rebuilding the desktop has the possibly undesirable side-effect of destroying Get Info comments you may have added to any programs or files.

Giving your program its own unique icon, rather than the generic one issued to new programs that don't have a 'BNDL' resource, is an easy way to add polish to your final application.

Chapter Program: Using Resources

This chapter's program, *ResourceUser*, does the following to demonstrate each of this chapter's topics:

- Retrieves a string from a 'STR#' resource and uses it to set a window's title.
- Retrieves a second string and uses it to draw text to a window.
- Loads 'PICT's and displays them in a loop to create animation.
- Plays three sounds on the Mac's speaker using 'snd ' resources.

Figure 3–24 shows the window you'll see when the program is complete.

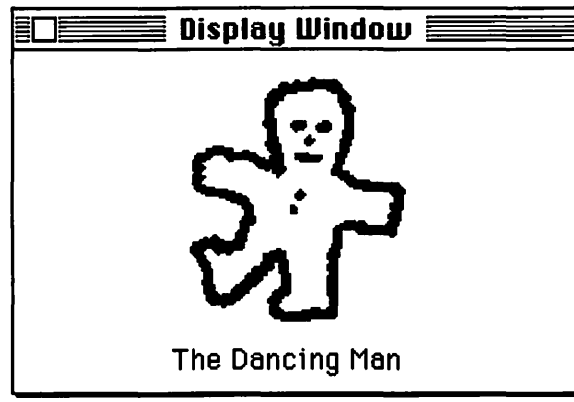


Figure 3–24. *The ResourceUser window*

When the sounds are done playing, click the mouse to end the program.

Program resources: *ResourceUser.π.rsrc*

ResourceUser's resource file contains the same 'WIND' resource found in the example programs of Chapters 1 and 2. It also contains 'STR#', 'PICT', and 'snd ' and 'BNDL' resources. You've already seen each of these resources scattered throughout this chapter. To save you the effort of going back through the pages I'll repeat them here as Figures 3–25, 3–26, 3–27, and 3–28.

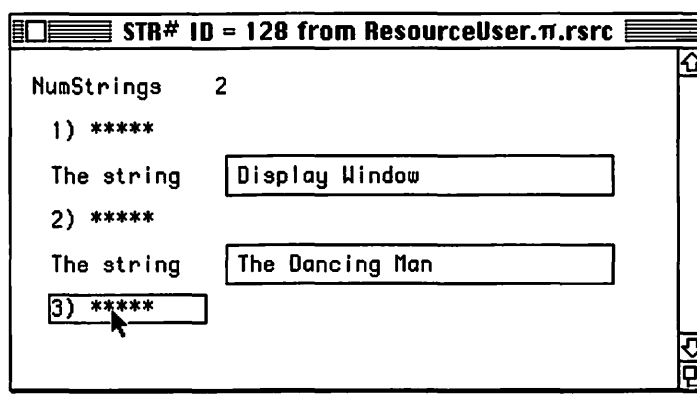


Figure 3–25. ResourceUser's *'STR#' resource*

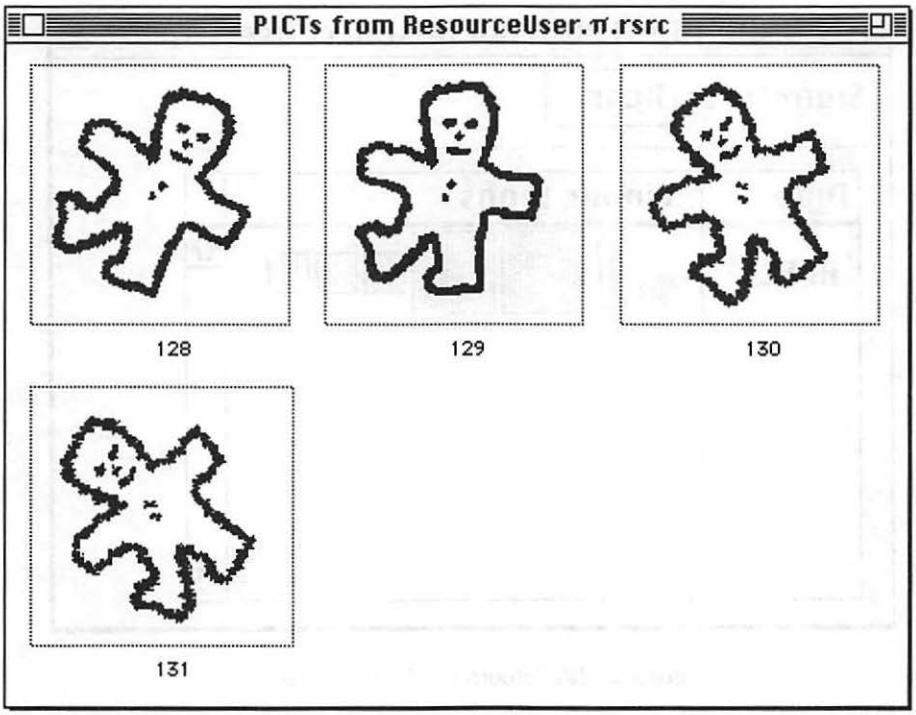


Figure 3-26. ResourceUser's *'PICT' resources*

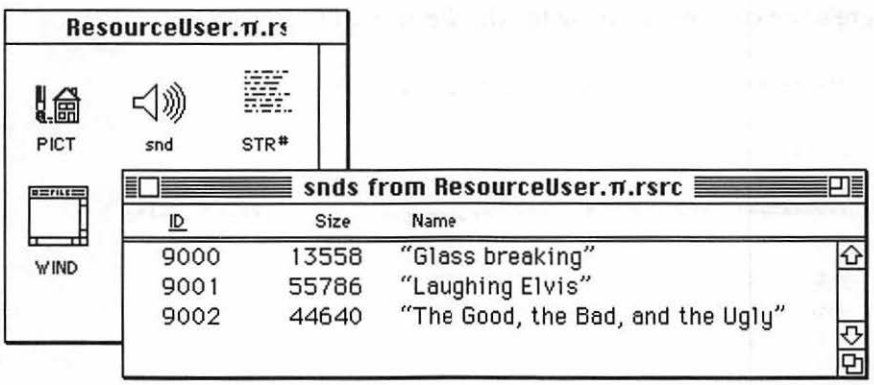


Figure 3-27. ResourceUser's *'snd ' resources*

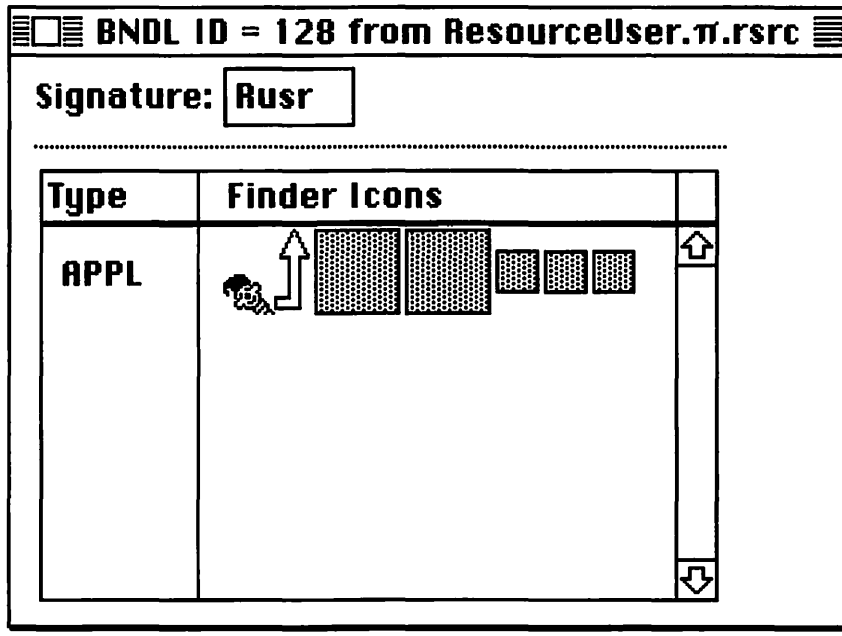


Figure 3–28. ResourceUser's *'BNDL' resource*

Program listing: *ResourceUser.c*

Here's the complete listing for the *ResourceUser* program.

```
/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++*/

#include <Sound.h>

/*++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void  Get_Some_Strings( void );
void  Draw_Moving_Picture( void );
void  Play_A_Sound( short );


/*++++++++++++   Define global constants   ++++++++++++++++++*/

#define    WIND_ID                 128
#define    NIL                      0L
#define    IN_FRONT     (WindowPtr)-1L
```

```
#define     REMOVE_EVENTS                0

#define     STR_LIST_ID                128
#define     WIND_TITLE_STR               1
#define     PICT_LABEL_STR               2
#define     PICT_LABEL_STR_L            70
#define     PICT_LABEL_STR_B           140

#define     FIRST_MAN_PICT             128
#define     PICT_L                      70
#define     PICT_T                      10
#define     DELAY_TICKS                  7

#define     SND_GLASS_ID              9000
#define     SND_ELVIS_ID              9001
#define     SND_CLINT_ID              9002


/*++++++++++++++   Define global variables   +++++++++++++++++*/

WindowPtr    The_Window;
Boolean      All_Done = FALSE;
EventRecord  The_Event;


/*++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();

   The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

   if ( The_Window == NIL )
      ExitToShell();

   SetPort( The_Window );
```

```
   Get_Some_Strings();

   Draw_Moving_Picture();

   Play_A_Sound( SND_GLASS_ID );
   Play_A_Sound( SND_ELVIS_ID );
   Play_A_Sound( SND_CLINT_ID );

   while ( All_Done == FALSE )
   {
      GetNextEvent( everyEvent, &The_Event );

      switch ( The_Event.what )
      {
         case mouseDown:
            All_Done = TRUE;
            break;
      }
   }
}


/*++++++++++++  Get strings from 'STR#' resource  ++++++++++++*/

void  Get_Some_Strings( void )
{
   Str255  the_str;

   SetPort( The_Window );

   GetIndString( the_str, STR_LIST_ID, WIND_TITLE_STR );
   SetWTitle( The_Window, the_str );

   GetIndString( the_str, STR_LIST_ID, PICT_LABEL_STR );
   MoveTo( PICT_LABEL_STR_L, PICT_LABEL_STR_B );
   DrawString( the_str );
}


/*++++++++++++  Create animation using 'PICT's  ++++++++++++++*/

void  Draw_Moving_Picture( void )
{
   Rect        pict_rect;
   PicHandle   pict_handle;
```

```
short       pict_wd;
short       pict_ht;
short       i;
short       count;
short       pict_id;
long        end_tick;

SetPort( The_Window );

pict_handle = GetPicture( FIRST_MAN_PICT );

pict_rect = ( **( pict_handle ) ).picFrame;

pict_wd  = pict_rect.right - pict_rect.left;
pict_ht  = pict_rect.bottom - pict_rect.top;

SetRect(&pict_rect, PICT_L, PICT_T, PICT_L + pict_wd, PICT_T + pict_ht);

count = 0;
for ( i=1; i < 31; i++ )
{
   ++count;
   switch ( count )
   {
      case 1:
         pict_id = FIRST_MAN_PICT;
         break;
      case 2:
         pict_id = FIRST_MAN_PICT + 1;
         break;
      case 6:
         pict_id = FIRST_MAN_PICT + 1;
         count = 0;
         break;
      case 3:
      case 5:
         pict_id = FIRST_MAN_PICT + 2;
         break;
      case 4:
         pict_id = FIRST_MAN_PICT + 3;
         break;
   }

   pict_handle = GetPicture( pict_id );
   DrawPicture( pict_handle, &pict_rect );
```

```
        Delay( DELAY_TICKS, &end_tick );
    }
}


/*+++++++++  Play one sound from a 'snd '  resource  +++++++++*/

void  Play_A_Sound( short snd_id )
{
    Handle  snd_handle;
    OSErr   err;

    snd_handle = GetResource( 'snd ' , snd_id );

    if ( snd_handle == NIL )
        ExitToShell();

    err = SndPlay( NIL, snd_handle, TRUE );

    if ( err != noErr )
        ExitToShell();
}
```

Stepping through the code

By now you're familiar with how the *main()* function of a Macintosh program works. Let's take a quick look at the #define directives, then concentrate on the three new routines: *Get_Some_Strings()*, *Draw_Moving_Picture()*, and *Play_A_Sound()*.

The #include directives

ResourceUser is the first program that uses an *#include* file. You need *Sound.h* because you will use the Sound Manager's *SndPlay()* routine.

```
#include <Sound.h>
```

The #define directives

You've seen *ResourceUser's* first four *#define* directives in previous programs. *WIND_ID* is the ID of the 'WIND' resource template. *NIL* and

IN_FRONT are parameters for *GetNewWindow()*. You'll use *REMOVE_EVENTS* at initialization.

The window's title and a string that's displayed in the window are in a string list resource. *WIND_TITLE_STR* and *PICT_LABEL_STR* each serve as an index to a string in the 'STR#' resource with an ID of *STR_LIST_ID*. *PICT_LABEL_STR_L* and *PICT_LABEL_STR_B* are pixel values that indicate where in the window the string will be drawn.

ResourceUser has four 'PICT' resources. The first is *FIRST_MAN_PICT*. You'll use *PICT_L* and *PICT_T* as pixel coordinates for drawing the picture in the window. *DELAY_TICKS* is used to slow down the animation.

ResourceUser plays three sounds. *SND_GLASS_ID*, *SND_ELVIS_ID*, and *SND_CLINT_ID* are the IDs of the three 'snd ' resources.

```
#define    WIND_ID                  128
#define    NIL                       0L
#define    IN_FRONT     (WindowPtr)-1L
#define    REMOVE_EVENTS             0

#define    STR_LIST_ID              128
#define    WIND_TITLE_STR             1
#define    PICT_LABEL_STR             2
#define    PICT_LABEL_STR_L          70
#define    PICT_LABEL_STR_B         140

#define    FIRST_MAN_PICT           128
#define    PICT_L                    70
#define    PICT_T                    10
#define    DELAY_TICKS                7

#define   SND_GLASS_ID             9000
#define   SND_ELVIS_ID             9001
#define   SND_CLINT_ID             9002
```

The main() function

ResourceUser borrows much of its *main()* function from past programs. I only discuss the differences here.

After initialization the program opens one window. A check is made to verify that the 'WIND' resource was properly loaded:

```
The_Window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

if ( The_Window == NIL )
   ExitToShell();
```

The program calls *Get_Some_Strings()* to load two strings. It then calls *Draw_Moving_Picture()* to display a dancing man to the window. The *Play_A_Sound()* routine is called three times, with a different 'snd ' resource ID passed each time.

```
Get_Some_Strings();

Draw_Moving_Picture();

Play_A_Sound( SND_GLASS_ID );
Play_A_Sound( SND_ELVIS_ID );
Play_A_Sound( SND_CLINT_ID );
```

Using strings

The *Get_Some_Strings()* routine relies on *GetIndString()* to load and display strings from a 'STR#' list. It uses the first string in the list, *WIND_TITLE_STR*, in a call to *SetWTitle()* to set the window's title to "Display Window."

```
void  Get_Some_Strings( void )
{
   Str255  the_str;

   SetPort( The_Window );

   GetIndString( the_str, STR_LIST_ID, WIND_TITLE_STR );
   SetWTitle( The_Window, the_str );

   GetIndString( the_str, STR_LIST_ID, PICT_LABEL_STR );
   MoveTo( PICT_LABEL_STR_L, PICT_LABEL_STR_B );
   DrawString( the_str );
}
```

Using pictures for animation

ResourceUser creates animation exactly as described earlier in this chapter. The *Draw_Moving_Picture()* routine was developed and explained in depth in this chapter's Animation Source Code section.

Playing sounds

After displaying the short animated sequence of pictures *ResourceUser* plays three sounds. You'll hear a glass shattering, Elvis Presley laughing, and a short sound clip from the Clint Eastwood movie "The Good, the Bad, and the Ugly."

Pass *Play_A_Sound()* a 'snd ' resource id and the routine will call *GetResource()* to load it into memory and *SndPlay()* to actually play it. If the sound is too large for the available memory, or if the sound doesn't play properly, the program will call *ExitToShell()* to terminate.

```
void  Play_A_Sound( short snd_id )
{
   Handle  snd_handle;
   OSErr   err;

   snd_handle = GetResource( 'snd ' , snd_id );

   if ( snd_handle == NIL )
      ExitToShell();

   err = SndPlay( NIL, snd_handle, TRUE );

   if ( err != noErr )
      ExitToShell();
}
```

Chapter Summary

Everything you see on the Macintosh screen was put there through the use of resources. A resource defines one element of the interface, such as a menu or window. Resources provide Macintosh programs with a uniform look.

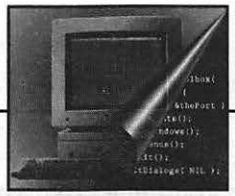
You use a resource editor like Apple's ResEdit to visually define the features of your program's windows, dialogs, menus, and alerts in a resource file. ResEdit can create and edit any of the 100 or so different resource types. A simple Macintosh application will use only a few of these types.

You can store lists of strings of text as resources using the 'STR#' resource type. You then use *GetIndString()* in your source code to load a string into memory. A call to *DrawString()* then displays it. This method of storing strings keeps text independent of your source code. That means others, with the use of a resource editor like ResEdit, can copy or edit the strings easily.

Pictures can be stored as resources using the 'PICT' resource type. A call to *GetPicture()* and *DrawPicture()* then loads and displays one picture. You can create 'PICT' resources by pasting any drawing made in a Macintosh drawing application into ResEdit.

The Macintosh is a multimedia machine. You can take advantage of its sound capabilities by storing sounds in 'snd ' resources. A call to *GetResource()* and the *SndPlay()* will load your sound into memory and play it on the speaker of you Macintosh.

Giving a program its own icon to be displayed in the Finder is the finishing touch that makes your program look professional. The 'BNDL' resource allows you to create your own icon and associate it with a program you write.



4 QuickDraw Graphics

What would be the point of programming on a Macintosh if you couldn't draw? Drawing is fun, creative, and gives you a chance to express yourself—something you can't say about some other areas of programming. If you're fortunate enough to have a color system, you can really let loose. This chapter will show you how.

Here you'll learn just what QuickDraw is and how it works. You'll also look at graphics ports, the data structures that allow drawing styles to change from one window to the next.

In this chapter you'll see how to draw lines and shapes. You'll then add a little flair to your shapes by filling them with patterns. Finally, you'll see how to add even more interest to your drawing by using color.

About QuickDraw

Everything you see on a Macintosh screen is there because of QuickDraw. QuickDraw is a group of Toolbox routines and is the single largest group of Toolbox functions. QuickDraw consists of more functions than any of the managers mentioned in Chapter 1.

Some things are obviously graphical, like the screen results of a paint program. But even windows, menus, and icons are all graphical images that have to be drawn. QuickDraw does this drawing. If any drawing has to be done, the managers rely on QuickDraw to do it.

While the managers indirectly make use of QuickDraw, you can directly use it by calling any of the hundreds of QuickDraw Toolbox functions.

If you're used to programming in a non-GUI environment, you might have written a few drawing routines of your own. Don't try bypassing QuickDraw by using or modifying any of your own routines. QuickDraw is fast, refined, and simple to use—you won't one-up it.

Initializing QuickDraw

QuickDraw has its own set of variables and data structures that need initialization. You've seen the following call in each program in this book:

```
InitGraf( &thePort );
```

Your program must make this call to initialize QuickDraw before any QuickDraw operations take place. Make this call right off the bat.

Speaking of initializations, you may recall that several other calls are included along with *InitGraf()*. They initialize other parts of the Toolbox, such as the Font Manager and the Window Manager. Note that the order in which these calls take place is extremely important and should remain the same as it is here.

To keep things nice and tidy I've combined all of these initialization calls into one function called *Initialize_Toolbox()*. You'll see this function called in the remaining example programs in this book. Here it is.

```
void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
```

```
    InitMenus();
    TEInit();
    InitDialogs( NIL );
    FlushEvents( everyEvent, REMOVE_EVENTS );
    InitCursor();
}
```

Pixels and the coordinate system

Chapter 1 introduced the pixel and the coordinate system. Remember from that discussion that the Macintosh uses bit-mapped graphics; every *pixel* on the screen has one or more bits in memory that keep track of the state of that pixel. For a monochrome Mac the state is on or off. For a color system, the state is the color of the pixel.

You can refer to each pixel by a pair of coordinates which define a point. This coordinate system starts at point (0, 0) in the upper-left corner of the screen and moves positively to the right and downward. Figure 4–1 is from Chapter 1. To illustrate the coordinate system, I've added the coordinates for a couple of pixels.

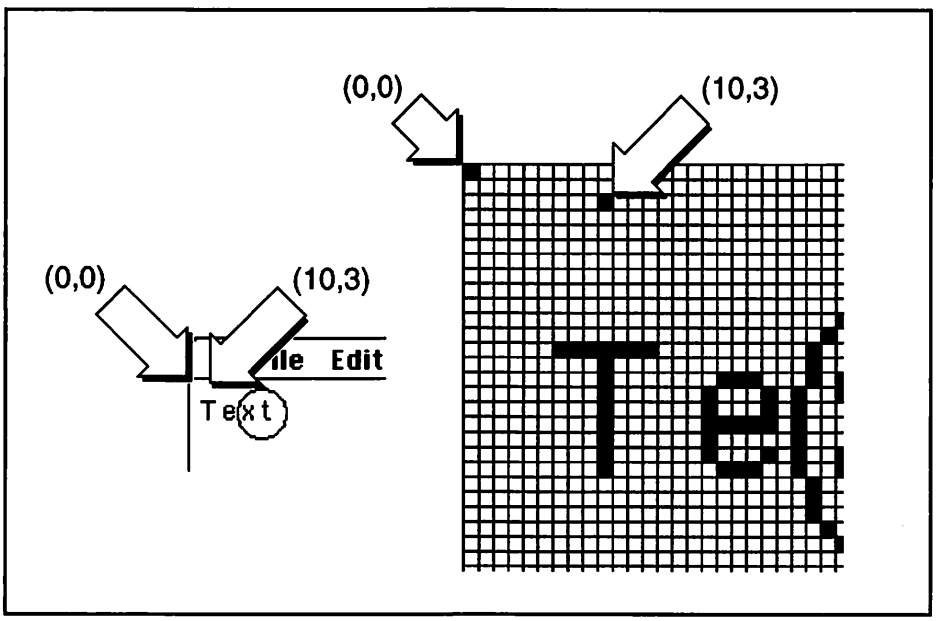


Figure 4–1. *The coordinate system*

The screen isn't the only part of the Macintosh that has a coordinate system. As you'll see in the very next section, every window on the screen has its own system.

Graphics Ports

When two windows are open on the screen, each is capable of displaying different styles of text. This is possible because each window has its own set of properties independent of all other windows.

The *GrafPort* and *GrafPtr*

Associated with a window is a *graphics port*. The port is the environment of the window. It describes the window's type and style of text, the thickness of drawn lines, and numerous other aspects of the graphics that go into the window.

With more than one window open on the screen you'll have to tell QuickDraw in which window or, more precisely, in which graphics port it should perform drawing operations. Issuing a call to *SetPort()* does this. *SetPort()* requires a pointer—a *GrafPtr*—to the port you wish to make the current port. A *GrafPort* is the structure that holds all this port information. A *GrafPtr* is a pointer to a *GrafPort*.

In previous chapters you've seen *SetPort()* in action in code that looks like this:

```
WindowPtr  The_Window;

SetPort( The_Window );
MoveTo( 30, 50 );
DrawString( "\pChapter One Program" );
```

You may wonder how I got away with passing *SetPort()* a variable of type *WindowPtr* when you now know that *SetPort()* requires a *GrafPtr*. Figure 4–2 hints at the answer. A *GrafPtr* points to a *GrafPort*. A *WindowPtr* points to a *WindowRecord* structure. Within the *WindowRecord*, the very first member is a *GrafPort*. So the first thing that both a *GrafPtr* and

a *WindowPtr* point to is a *GrafPort* which is good enough for *SetPort()* and good enough for us. You'll learn all the sordid details about *WindowRecords* in the next chapter.

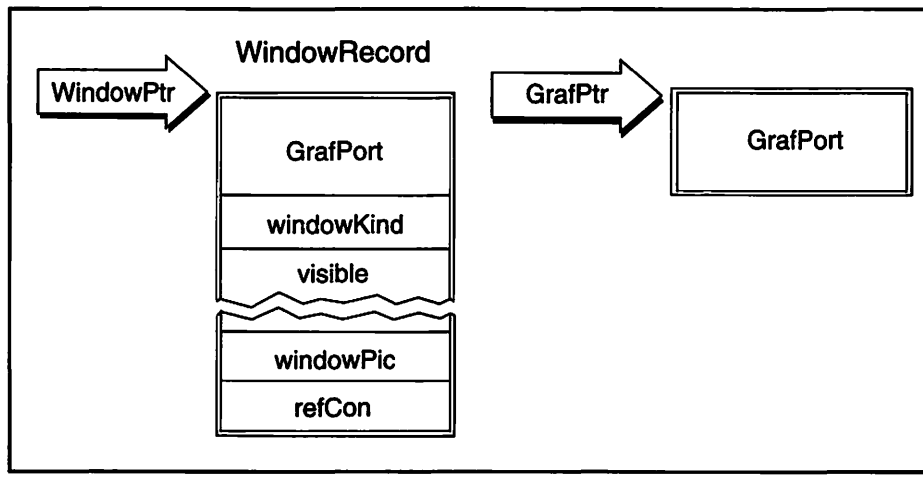


Figure 4–2. *The WindowPtr and GrafPtr*

The graphics pen

A graphics port holds the graphical information about a port. When you draw to a window, QuickDraw uses the information held in that window's graphics port. By adjusting the settings of the *graphics pen* you can change many of the port's drawing properties. The graphics pen is an invisible drawing tool that exists as a convenience for making changes to the properties of lines drawn in a window.

You saw the pen in use in the example program of Chapter 1 with the call to the Toolbox routine *MoveTo()*. *MoveTo()* moves the pen, without drawing, to the pixel coordinates you specify. The reference point for moving is the window's upper left corner. The companion function to *MoveTo()* is *Move()*. *Move()* uses the pen's current position as a reference, not the window's corner. Figure 4–3 shows where the pen would end up after a call to *MoveTo(150, 100)*. Figure 4–3 also shows that each port has its own coordinate system. Don't forget, each window has a port, and the screen itself is a port.

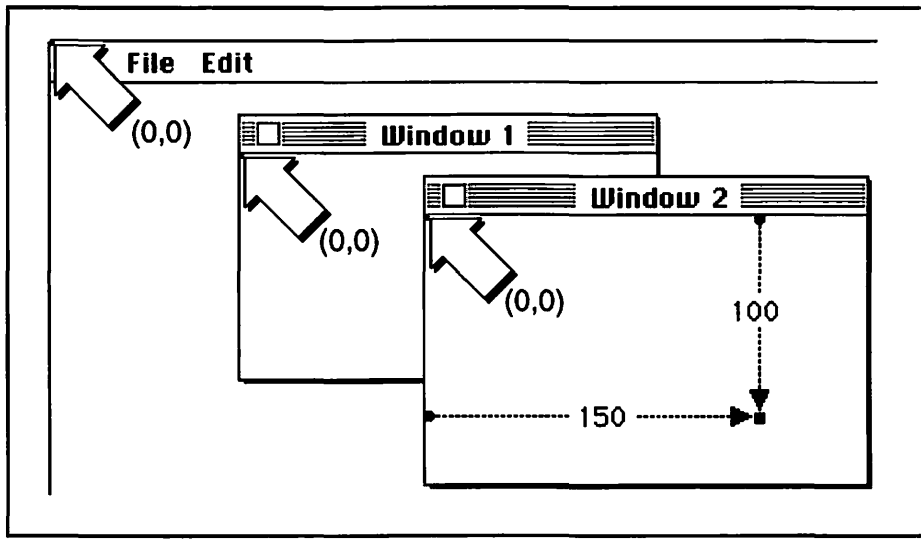


Figure 4–3. *Result of* MoveTo(150,100) *in a window's port*

You just saw that you can move the pen. You can also change its characteristics. Call *PenSize()* to change the size of the pen's tip. The first parameter to *PenSize()* controls the pen's height, the second parameter controls the pen's width.

Changing the pen size will affect the thickness of lines drawn with all subsequent calls to *LineTo()*. The first parameter to *LineTo()* gives the horizontal length of a line and the second parameter gives the vertical length. The reference point for the line is the window's upper-left corner. The companion to *LineTo()* is *Line()*, which uses the current location of the pen as its reference. Here's a code fragment using all five of these calls.

```
PenSize( 1, 3 );
MoveTo( 100, 100 );
Line( 90, -50 );
Move( 100, 0 );
LineTo( 290, 140 );
```

Figure 4–4 shows the results of the above code. Note that a negative vertical value sends the pen upward. For the horizontal direction a negative value would move the pen to the left.

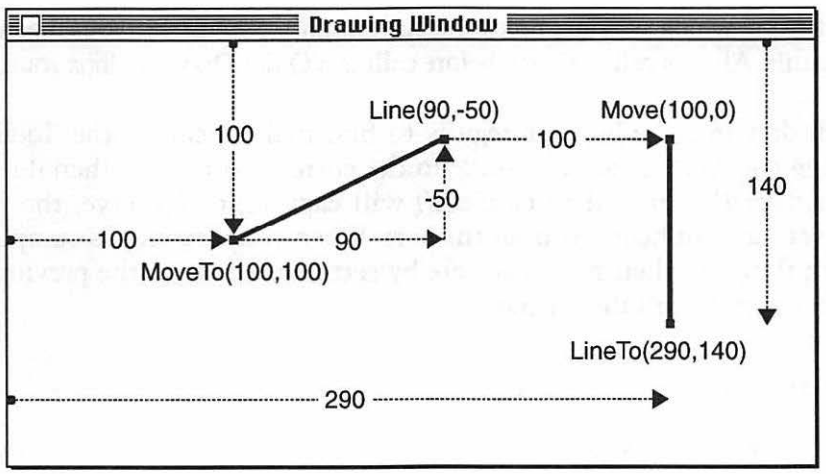
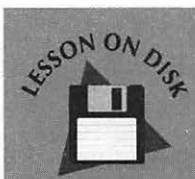


Figure 4–4. *Results of moving and line drawing*



Lesson 4–1: Moving the Pen

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Defensive Drawing

Every window has its own port, and this allows a program's user to switch back and forth between these windows at will. Many applications allow the user to select different graphics settings in each window. It's not up to the user to keep track of all of this; it's up to you, the programmer. Fortunately, the Toolbox contains a few routines that make this task painless.

Changing ports

When you issue a command to QuickDraw, it will faithfully execute that command. The results of the command will always end up in the current port. If you have more than one window on the screen, you must tell

QuickDraw which window holds the current port. *SetPort()* is your means of doing this. Always call *SetPort()* before calling a QuickDraw Toolbox routine.

When drawing, the best strategy is to first make a call to the Toolbox routine *GetPort()* to get a *GrafPtr* to the current port. Only then do you call *SetPort()*. The call to *GetPort()* will capture, or preserve, the port that was current before you set the port. When your drawing is complete, return things to their previous state by setting the port to the previously current port. Here's the format.

```
void  Draw_Something( GrafPtr draw_port )
{
   GrafPtr    save_port;

   GetPort( &save_port );
   SetPort( draw_port );

   [ perform drawing here ]

   SetPort( save_port );
}
```

Notice that in the C language the *GetPort()* routine accepts a pointer to *GrafPtr* as a parameter while *SetPort()* accepts a *GrafPtr*.

Note that the *GrafPtr* that is passed to *Draw_Something()* can also be a *WindowPtr*:

```
void  Draw_Something( WindowPtr  draw_window )
{
   GrafPtr    save_port;

   GetPort( &save_port );
   SetPort( draw_window );

   [ perform drawing here ]

   SetPort( save_port );
}
```

Apple states that the misuse of *SetPort()* is one of the most common sources of errors in programming the Macintosh. Don't ignore Apple!

Even if your application uses only one window you should still adhere to this strategy. If the user opens a desk accessory while your program is running, the output of your program could end up in the desk accessory's window. Desk accessory windows are ports, too!

Lesson 4–2: Switching Ports

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Changing characteristics of a port

One of the reasons the Macintosh gained its reputation as a computer that is easy to use is because the Mac gave control to the user. Program users don't have to be programmers to change the look of text or to draw into windows. Macintosh applications let users make changes easily to a window's environment, or graphics port, through menu choices or dialog box selections.

When a user makes an effort to set graphics characteristics for a desired effect, that user will find it disconcerting if the characteristics change on their own. If you're going to change the state of the graphics pen, you'll want to first save the present state of the pen with *GetPenState()*. Pass *GetPenState()* a variable of type *PenState*. You can then change properties of the pen with calls to routines like *PenSize()*. When done, return the pen to its previous condition with a call to *SetPenState()*. Here's a code fragment that does that:

```
PenState    save_state;

GetPenState( &save_state );
```

```
[ change pen characteristics ]

SetPenState( &save_state );
```

When would a program allow both the user and the program itself to change the state of the pen? Figure 4–5 shows one possibility. In this hypothetical paint program, the user clicked on a line thickness of four to change the pen size. When the user drew a circle, it was drawn with the selected pen size. The program has a feature that automatically adds a cross hair to a circle always using a pen size of one pixel. After the cross hair is drawn the program should return the pen to the state the user last selected—a size of four pixels.

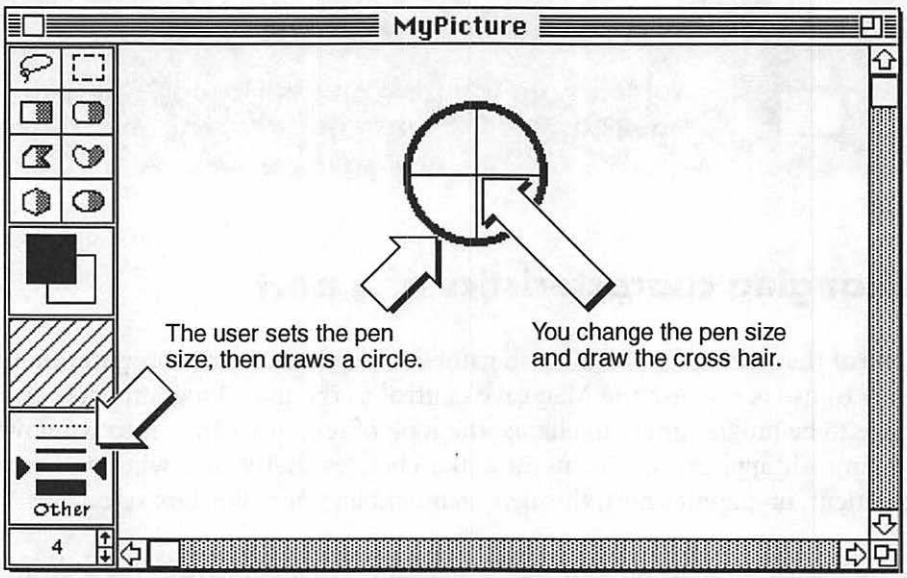


Figure 4–5. *Both the user and program can control the pen*

Let's summarize the defensive drawing tactics covered in this section:

- Save the current port with *GetPort()*.
- Make the port you're about to use the active port with *SetPort()*.
- Save the state of the graphics pen with *GetPenState()*.
- Make any desired pen changes.

- Draw any desired shapes.
- Reset the state of the pen with *SetPenState()*.
- Reset the port to the previously active port with *SetPort()*.

Let's end this section with a final version of *Draw_Something()*. You'll want to pattern all your routines that change the pen or draw to a window on this one. Keep in mind that the calls to these routines will add very little to the size of your final application and may save you hours in trying to find the cause of bugs later on.

```
void  Draw_Something( WindowPtr  draw_window )
{
   GrafPtr    save_port;
   PenState   save_state;

   GetPenState( &save_state );

   GetPort( &save_port );
   SetPort( draw_window );

   [ change pen characteristics ]
   [ perform drawing operations ]

   SetPenState( &save_state );

   SetPort( save_port );
}
```

Drawing Shapes

In Chapter 3 you saw the *Rect* data type and *SetRect()*, the Toolbox call that establishes the boundaries of a *Rect*. The rectangle is the basis of many of the shapes QuickDraw creates, so I'll repeat the last chapter's call to *SetRect()* and the figure that displayed the results. Here's the call. Figure 4–6 shows the results.

```
#define   LEFT    75
```

```
#define   TOP     40
#define   RIGHT   175
#define   BOTTOM  90

SetRect( &the_rect, LEFT, TOP, RIGHT, BOTTOM );
```

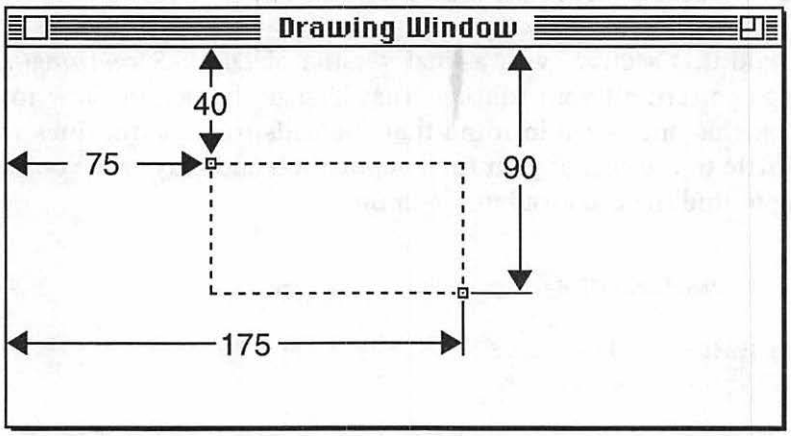


Figure 4–6. SetRect() *sets a rectangle's boundaries*

Once you've set the boundaries for a rectangle you can perform several different drawing operations on the rectangle, as discussed in the next section.

Working with rectangles

With the bounds of a rectangle established, you can frame it with *FrameRect()*:

```
Rect  the_rect;

FrameRect( &the_rect );
```

If you'd like to fill the inside of a rectangle with a pattern, you can use *FillRect()*. Pass *FillRect()* a pointer to the *Rect* to fill and the pattern to fill it. There are five standard patterns of the C data type *Pattern* available for your use: *white*, *ltGray*, *gray*, *dkGray*, and *black*. Keeping in mind that C is case-sensitive, use one of these patterns as the second parameter:

```
FillRect( &the_rect, ltGray );
```

Earlier I introduced the graphics pen. You saw that it could draw black lines using *Line()* and *LineTo()*. These lines don't have to be black. You can change the pattern that the pen uses with a call to *PenPat()*. Include one of the predefined patterns as the sole parameter. Here's a call that draws a diagonal line in a dark gray pattern rather than black.

```
PenPat( dkGray );
MoveTo( 20, 30 );
Line( 100, 100 );
```

Once you change the pen pattern, the change stays in effect until the next call to *PenPat()*. If the pen pattern is already to your liking, you can call *FillRect()*'s companion routine *PaintRect()*. The only difference between the two is that *PaintRect()* uses the current pen pattern to fill the rectangle, while *FillRect()* requires that you pass a pattern as a parameter.

```
Rect  the_rect;

SetRect( &the_rect, 20, 20, 120, 120 );
PenPat( gray );
PaintRect( &the_rect );
SetRect( &the_rect, 50, 50, 150, 150 );
FillRect( &the_rect, black );
```

Figure 4–7 shows the result of this code. Note that the call to *PaintRect()* uses the current pen pattern *gray*, as set by the call to *PenPat()*. *FillRect()* ignores the current pen pattern and uses the passed pattern of *black*. The next section discusses patterns in greater detail.

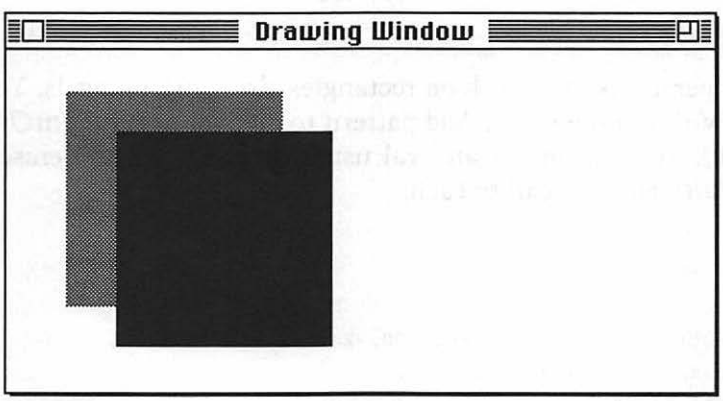


Figure 4–7. *Result of calling* PaintRect() *and* FillRect()

You can invert a rectangle using *InvertRect()*. This routine doesn't add a pattern to a rectangle like *PaintRect()* or *FillRect()*. Instead, it inverts each pixel that falls within the boundaries of the rectangle. If the window happens to be all white at the time of the call, the rectangle will be all black.

```
InvertRect( &the_rect )
```

When you're finished displaying a rectangle you can remove it with a call to *EraseRect()*, which will wipe out the entire rectangle and replace it with the background color, usually white.

```
EraseRect( &the_rect )
```

Working with ovals

Now that you know all about rectangles, ovals will be a breeze. An oval begins with a call to *SetRect()*. Why set a rectangle to draw an oval? QuickDraw will not display the rectangle; it will only use it as a guide in which to inscribe the oval when you call *FrameOval()*. Look at the following code, then check out the results in Figure 4–8. Take note that the dashed rectangle in Figure 4–8 is only there to give a feel for what bounds the oval; QuickDraw will not actually display it.

```
Rect  the_rect;

SetRect( &the_rect, 50, 50, 200, 150 );
FrameOval( &the_rect );
```

All the operations that work on rectangles also work on ovals. You frame an oval with *FrameOval()*. Add pattern to an oval using *PaintOval()* and *FillOval()*. You can invert an oval using *InvertOval()* and erase it with *EraseOval()*. Here's a call to each:

```
Rect  the_rect;

SetRect( &the_rect, 60, 80, 200, 235 );
FillOval( &the_rect, dkGray );
PenPat( black );
SetRect( &the_rect, 150, 180, 300, 330 );
```

```
PaintOval( &the_rect );
SetRect( &the_rect, 100, 100, 160, 185 );
InvertOval( &the_rect );
SetRect( &the_rect, 200, 200, 250, 250 );
EraseOval( &the_rect );
```

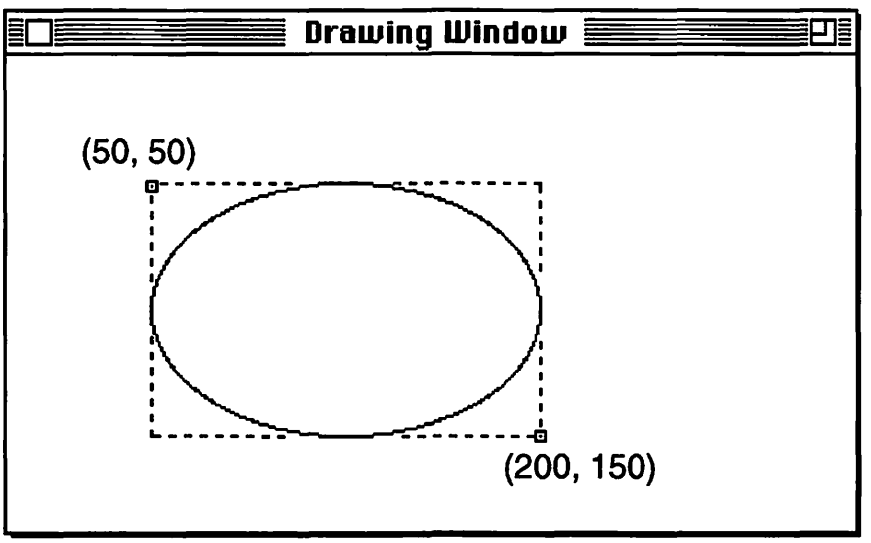


Figure 4-8. *An oval is inscribed in the boundaries set by SetRect()*

Working with round rectangles

The Macintosh has an interesting shape called the round rectangle, which is a rectangle with rounded-off edges. If you think back to the definition of an oval, you'll have a pretty good clue of how the Macintosh defines the round rectangle.

```
short  wd = 100;
short  ht =  50;
Rect   the_rect;

SetRect( &the_rect, 40, 60, 240, 160 );
FrameRoundRect( &the_rect, wd, ht );
```

First, set the boundary rectangle with *SetRect()*. Then define the pixel width and height of an imaginary oval that defines the degree of rounding of the corners. QuickDraw uses this oval for rounding each corner.

Pass the oval width and height to *FrameRoundRect()*. Figure 4.9 illustrates the results of the above code fragment.

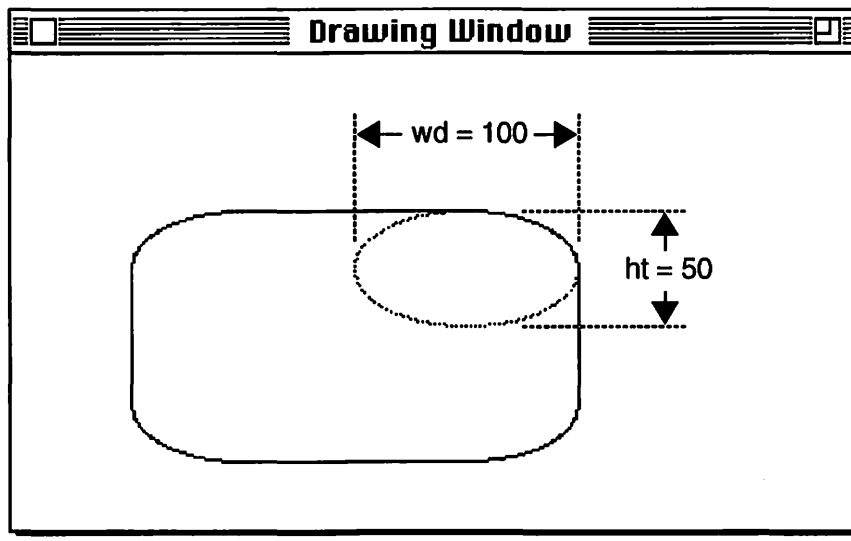


Figure 4–9. *An oval defines the corners of a round rectangle*

Don't be surprised to learn that round rectangles can have the same operations performed on them as rectangles and ovals. Frame a round rectangle with *FrameRoundRect()*. Apply a pattern to a round rectangle using *PaintRoundRect()* or *FillRoundRect()*. Invert a round rectangle using *InvertRoundRect()*. Finally, erase a round rectangle using *EraseRoundRect()*. Once again, here is a call to each:

```
short  wd = 40;
short  ht = 75;
Rect   the_rect;

SetRect( &the_rect, 10, 10, 200, 200 );
FillRoundRect( &the_rect, dkGray );
PenPat( ltGray );
SetRect( &the_rect, 30, 200, 100, 250 );
PaintRoundRect( &the_rect );
SetRect( &the_rect, 50, 45, 255, 320 );
InvertRoundRect( &the_rect );
SetRect( &the_rect, 200, 100, 250, 250 );
EraseRoundRect( &the_rect );
```

Patterns

The five standard patterns are handy to have around, but you'll find occasion to develop your own. That's easy to do with the aid of the 'PAT ' resource type. You'll use ResEdit to edit your own pattern and then a little C source code to make it work for you.

Remember, every resource type has a four-character name. For the 'PAT ' type, there is a space after the 'T'.

The 'PAT ' Resource

Since you're proficient with ResEdit by now, you will know how to create a pattern resource by simply looking at the following.

1. Choose "Create New Resource" from the Resource menu.

2. In the Select New Type dialog that opens, double-click on 'PAT ' in the list of resource types.

3. A pattern editor will open. There, click the small pencil on individual pixels on the left side of the window. A larger section of pattern will be shown on the right side. Figure 4–10 illustrates the procedure.

You'll edit an 8-pixel-by-8-pixel square. Later, when your program uses this pattern, QuickDraw will lay copies of that square end-to-end and side-by-side to fill whatever area you specify.

Figure 4–11 shows a completed 'PAT ' resource. In the next section you'll call on this resource to fill the lines and shapes that you display with QuickDraw calls.

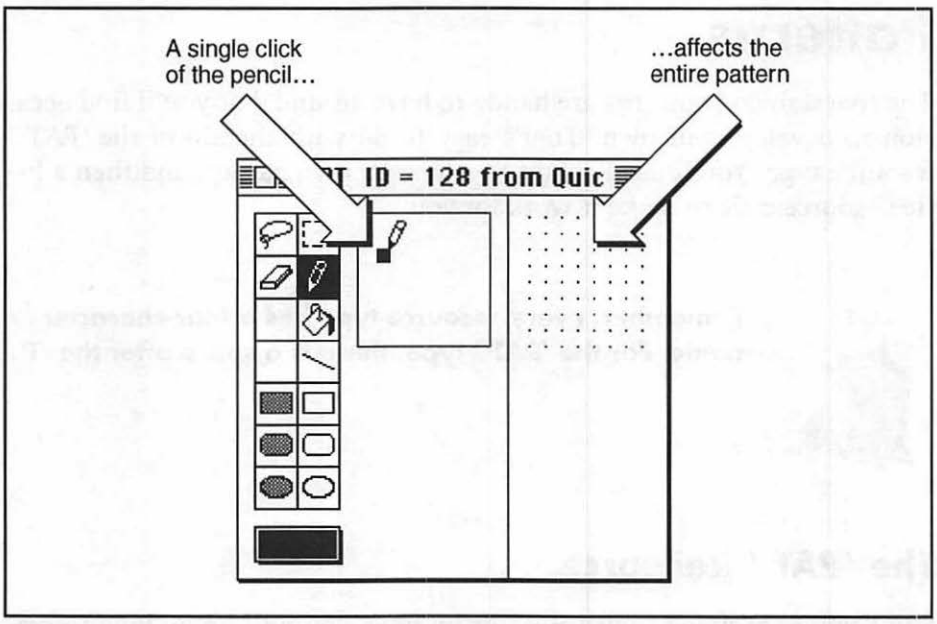


Figure 4–10. *ResEdit's pattern editor*

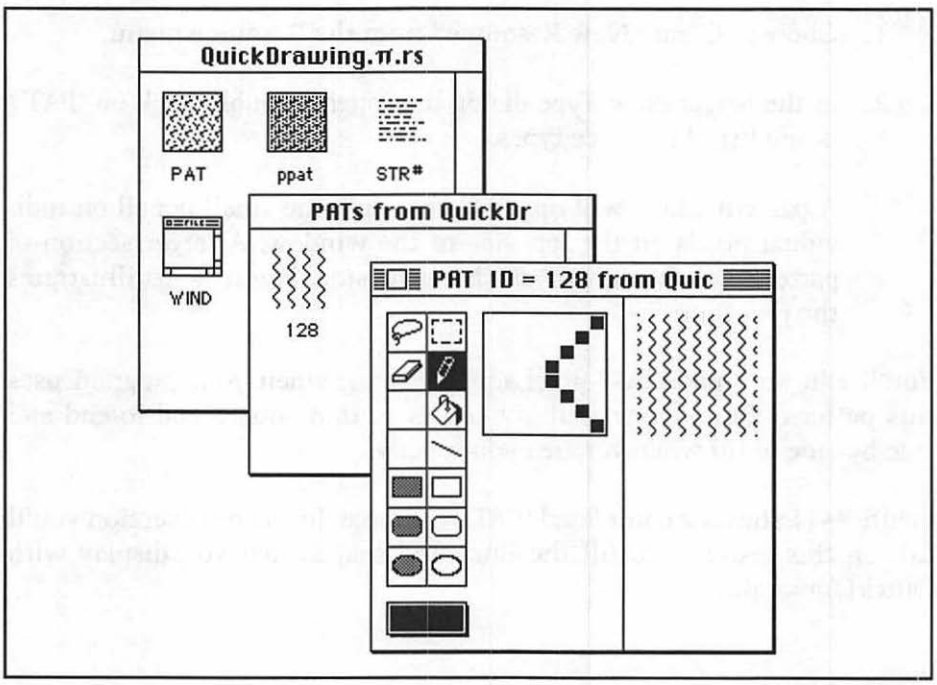


Figure 4–11. *A 'PAT ' in the pattern editor*

The pattern source code

By now you should be able to see the pattern for using resources—no pun intended. First, you use ResEdit to create the appropriate resource. Then you use a Toolbox call to load that resource into memory. The Toolbox gives your program a handle to the resource. That gives you something to work with.

Patterns follow this same process. You created a 'PAT ' resource in ResEdit. Now, bring it into memory with a call to *GetPattern()*. Pass *GetPattern()* the resource ID of the 'PAT ' to load. In return, *GetPattern()* will give your program a handle to the pattern in memory. Not just an ordinary handle, of course. You'll get a *PatHandle*. Here's how it's done:

```
#define    MY_PAT_ID        128

PatHandle  pen_pat_handle;

pen_pat_handle = GetPattern( MY_PAT_ID );
```

What can you do with the handle? By dereferencing the handle twice you move from a pattern handle to a pattern pointer, then to a *Pattern*. Note the capital 'P' in *Pattern*. When speaking of patterns in general, use lowercase. When referring specifically to the Macintosh C data type, use *Pattern*. You can pass a *Pattern*, or a doubly-dereferenced handle, to *PenPat()* to change the current pattern of the pen. Then, any drawing that you do, whether it be lines or shapes, will make use of your new pattern. Here's a comprehensive example. Figure 4–12 follows and shows the result.

```
#define    MY_PAT_ID        128

PatHandle  pen_pat_handle;
Rect       the_rect;

pen_pat_handle = GetPattern( MY_PAT_ID );
PenPat(**pen_pat_handle);

PenSize( 10, 10 );
MoveTo( 20, 20 );
Line( 300, 0 );

SetRect( &the_rect, 20, 50, 150, 100 );
PaintRect( &the_rect );
```

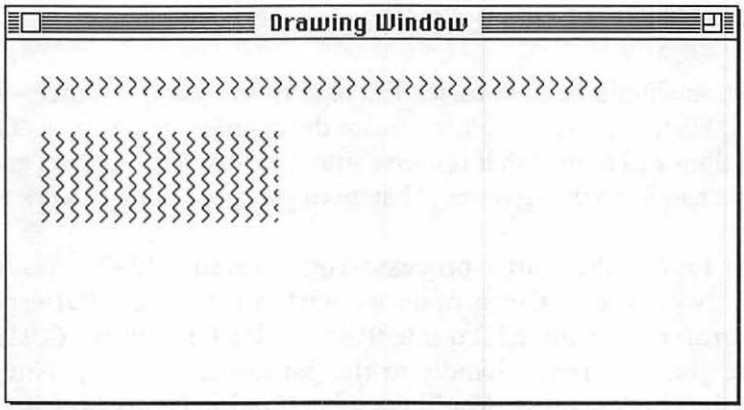


Figure 4–12. *Drawing routines using the 'PAT ' resource*

Creating a 'PAT ' resource is simple and fun. Using the resource in your source code is just as easy. Since the number of patterns you can develop is huge, the 'PAT ' resource can really open the door for you to express your own creativity.

Lesson 4–3: Using Patterns

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Color QuickDraw

If you're fortunate, you have a color Macintosh system. Even if you don't, you should know how to include color in your programs. You'll want program users who do have color to be able to take full advantage of their machines.

Checking for color

Whether or not the user of your program has a color system is an important issue. Users with older systems have the original QuickDraw in

their machines, not the newer Color QuickDraw. Toolbox routines you'll be calling to display color will fail on older machines—something you certainly want to avoid. Chapter 8 deals extensively with issues such as checking for color. Here, I'll just tell you a little about an essential check you must make.

The Toolbox routine *Gestalt()* checks the system a program is running on for a variety of things, including the presence of Color QuickDraw. *Gestalt()* is covered in great depth in Chapter 8. Here you need only know how to use it and not all the details on how it works.

Include the header file *GestaltEqu.h* in any program that might make use of Color QuickDraw. That will allow you full use of the *Gestalt()* function.

```
#include <GestaltEqu.h>
```

Next, declare a global *Boolean* variable that will be the flag that tells your program whether Color QuickDraw is present.

```
Boolean  Color_QD_Present;
```

Near the start of your program, call *Gestalt()* with the two parameters shown below.

```
OSErr  err;
long   response;

err = Gestalt( gestaltQuickdrawVersion, &response );
```

The rather ungainly constant *gestaltQuickdrawVersion* will tell the versatile *Gestalt()* that on this occasion it should check for the version of QuickDraw. *Gestalt()* is capable of checking for a number of other system parameters. *Gestalt()* will dig that information out of the Mac your program is running on and relay it to your program in the variable-named response. It will also notify your program if it somehow failed its mission; that's what the *err* variable is for.

Immediately after the call to *Gestalt()*, check the results. Make sure there was no error and then set the color QuickDraw flag according to the value held in response. A response value of *gestaltOriginalQD*

means this system has the original black and white version of QuickDraw. Any other value means there's one of several versions of Color QuickDraw present.

```
if ( ( err == noErr ) && ( response == gestaltOriginalQD ) )
   Color_QD_Present = FALSE:
else
   Color_QD_Present = TRUE:
```

Now, whenever your program is about to make a call to a Color QuickDraw routine that could spell disaster on a monochrome system, you can check your *Color_QD_Present* variable to see if it's safe to continue. And if it isn't? Then you'll use a similar call that works for black and white.

You'll see this color-checking code bundled together in the example program at the end of this chapter. And remember, you'll get more explanation when you arrive at Chapter 8.

Color windows

You've already used the *GetNewWindow()* routine several times. To make use of color in a window you'll call *GetNewCWindow()*. Both calls ask for the same number and type of parameters, and both return a *WindowPtr*. Here's a call to each.

```
#define    NIL                    0L
#define    IN_FRONT   (WindowPtr)-1L

WindowPtr  The_BW_Window;
WindowPtr  The_C_Window;

The_BW_Window = GetNewWindow( BW_WIND_ID, NIL, IN_FRONT );
The_C_Window  = GetNewCWindow( C_WIND_ID, NIL, IN_FRONT );
```

Earlier you read about graphics ports and the *GrafPort* type—every window has one. For drawing in color there's a special type of port called a *CGrafPort*. Standard windows have a *GrafPort*. Color windows have a *CGrafPort*. Most of the differences will be transparent to you. Drawing operations in either type of port are similar, as you'll soon see.

Before working with color you'll test your color flag variable to see if color is available. If it is, open a color window; if not, put the traditional monochrome window up. Here's how:

```
WindowPtr  The_Window;

if ( Color_QD_Present == TRUE )
   The_Window = GetNewCWindow( C_WIND_ID, NIL, IN_FRONT );
else
   The_Window = GetNewWindow( BW_WIND_ID, NIL, IN_FRONT );
```

If you know how to draw, you know how to draw in color. Everything covered up to this point has been for monochrome, but it applies to color as well. Use a color port just as you would a standard port. First use *SetPort()* to make it the current port, then use QuickDraw routines to draw to it. Here's an example that draws a line in a window.

```
WindowPtr  The_Window;

if ( Color_QD_Present == TRUE )
   The_Window = GetNewCWindow( C_WIND_ID, NIL, IN_FRONT );
else
   The_Window = GetNewWindow( BW_WIND_ID, NIL, IN_FRONT );

SetPort( The_Window );

MoveTo( 20, 20 );
Line( 100, 100 );
```

Of course, there's not much point in using color windows if you're not using a color monitor.

Color patterns

The monochrome representation of a pattern is the 8-pixel-by-8-pixel square with a C data type of *Pattern*. For color, the size is the same, but each pixel can take on any of the available colors, not just black or white. The C data type for a color pattern is *PixPat*, a pixel pattern. There's also a color pattern resource, the 'ppat'.

The 'ppat' resource

The color 'ppat' is analogous to the monochrome 'PAT '. If you know how to use ResEdit's pattern editor (and you should), then you already know how to make a 'ppat' resource. For a color pattern you select colors for each pixel. To do so, you use a palette that opens when you click on the rectangle in the lower-left corner of the pattern editor. Figure 4–13 shows a color pattern and the color selection palette—in black and white print, unfortunately—in ResEdit.

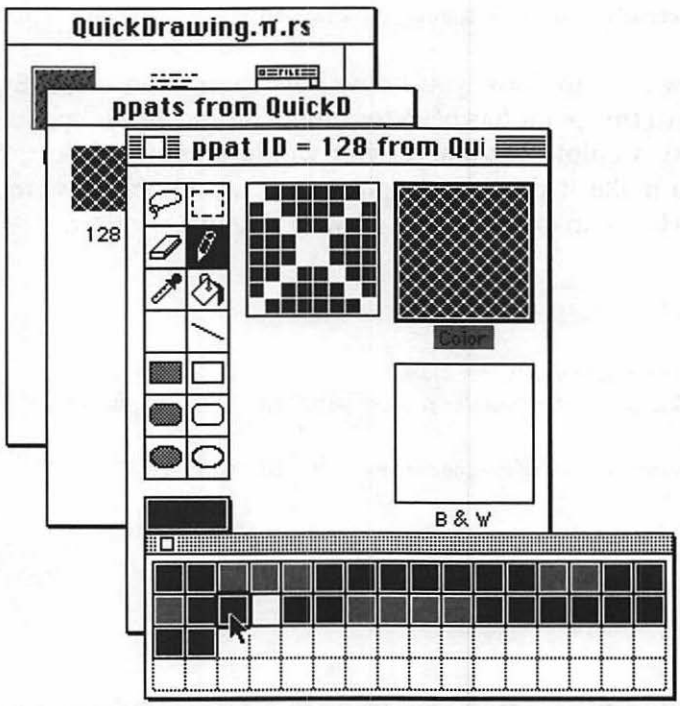


Figure 4–13. *A 'ppat' in the pattern editor*

The color pattern source code

Similar to *GetPattern()*, the call that brings a 'PAT ' into memory, *GetPixPat()* is the call that loads a 'ppat' into memory. *GetPixPat()* returns yet another handle type, a *PixPatHandle*.

```
#define    PEN_PAT_C_ID        128

PixPatHandle  pen_pixpat_handle;

pen_pixpat_handle = GetPixPat( PEN_PAT_C_ID );
```

To change the current setting of the pen to your new color pattern, use the color version of *PenPat()*; that is, *PenPixPat()*. This routine conveniently takes a *PixPatHandle* as its parameter, so there's no dereferencing involved. You have the handle from the call to *GetPixPat()*, now use it in *PenPixPat()*, as shown in this example:

```
#define    PEN_PAT_C_ID        128

PixPatHandle  pen_pixpat_handle;
Rect          the_rect;

pen_pixpat_handle = GetPixPat( PEN_PAT_C_ID );
PenPixPat( pen_pixpat_handle );

PenSize( 10, 10 );
MoveTo( 20, 20 );
Line( 300, 0 );

SetRect( &the_rect, 20, 50, 150, 100 );
PaintRect( &the_rect );
```

If the above example looks familiar, it should; the last five lines are the same as those of the monochrome pattern example a few pages back. Once the pen pattern is set—whether it be with a call to *PenPat()* or a call to *PenPixPat()*—line drawing and shape painting takes place with the same calls. Shape filling is just a little different, as you'll soon see.

Figure 4–14 shows the results you could expect from the above example, assuming the 'ppat' pattern shown in Figure 4–13 is used. Again, the actual pattern contains whatever colors were used for the 'ppat'.

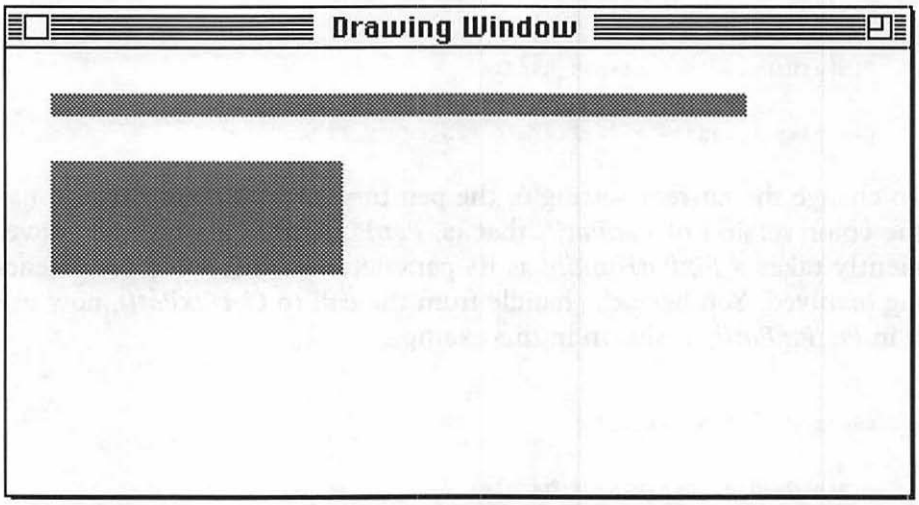


Figure 4–14. *Drawing routines use your 'PAT' resource*

Color drawing

Now that you know about color patterns, the rest of color is a snap. Everything you know from the "old" monochrome QuickDraw applies. Once you set the pen pattern using *PenPixPat()*, lines and painted shapes will use this new pattern. For instance, the preceding example used *Line()* and *PaintRect()*.

Fill routines, such as *FillRect()* require that you specify the pattern to use; it ignores the current pen pattern. With color you're working with a *PixPatHandle* and not a *Pattern*. Because of this the fill routines for color QuickDraw are somewhat different. Each of the monochrome Toolbox routines has a sister routine for color. Here's a call to each:

```
#define    PEN_PAT_C_ID        128

PixPatHandle  fill_ppat_handle;
Rect          the_rect;

fill_ppat_handle = GetPixPat( FILL_PAT_COLOR_ID );

SetRect( &the_rect, 20, 150, 200, 250 );
FillCRect( &the_rect, fill_ppat_handle );
```

```
FillCOval( &the_rect, fill_ppat_handle );
FillCRoundRect( &the_rect, fill_ppat_handle );
```

Inverting shapes in monochrome is simple because black is defined as the opposite of white. For color, things aren't quite so simple. Just what is the opposite of light chartreuse, anyway? It is possible to invert all or part of a color shape by calling *InvertRect()*, but you should avoid an inversion attempt such as this because of its unpredictable nature.

Toolbox routines originally intended for monochrome systems will work in color windows. The reverse is not always true. A call to *FillRect(&the_rect, ltGray)* will draw a light gray rectangle in a color window. A call to *FillCRect(&the_rect, fill_ppat_handle)* will not draw anything if color QuickDraw is not present.

Drawing in color isn't the only thing that adds color to a window. You can use ResEdit to add color to parts of a window's frame or content, too. That's next.

The 'wctb' resource

When you create a 'WIND' resource in ResEdit you do so in the window editor. There, you can add color to some of the parts of a window, such as its content.

Click the Custom button in the window editor to see the five parts of a window that are capable of displaying color. Then click on one of the five rectangles to open a palette of colors. ResEdit will reflect your color choice in the MiniWindow of the window editor. This is shown in Figure 4–15. Before the change is made, ResEdit will notify you that adding color to a 'WIND' adds a new resource to your resource file. ResEdit adds a window color table, a 'wctb' resource. Figure 4–16 shows the alert ResEdit puts up. Click OK to continue.

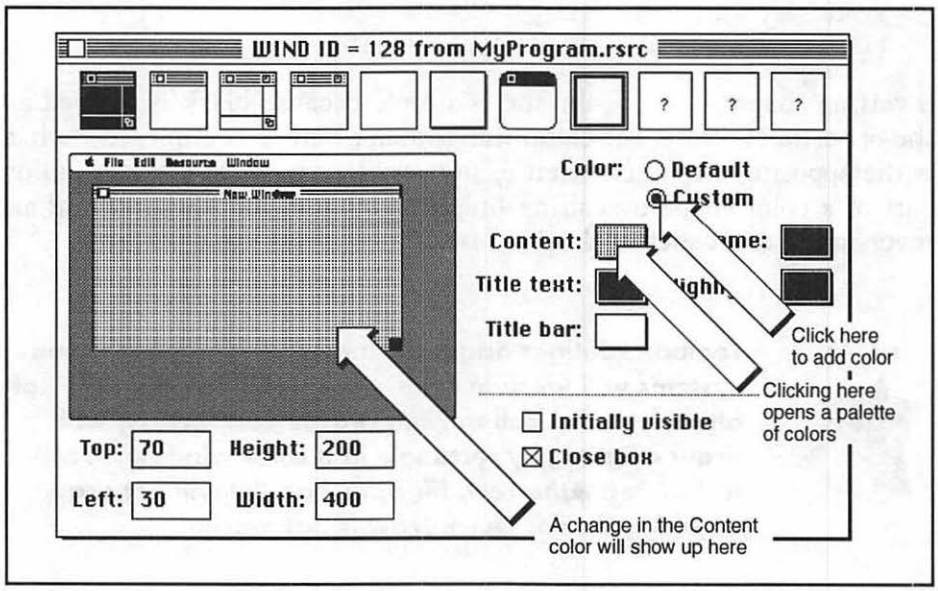


Figure 4–15. *Changing window colors in ResEdit's window editor*

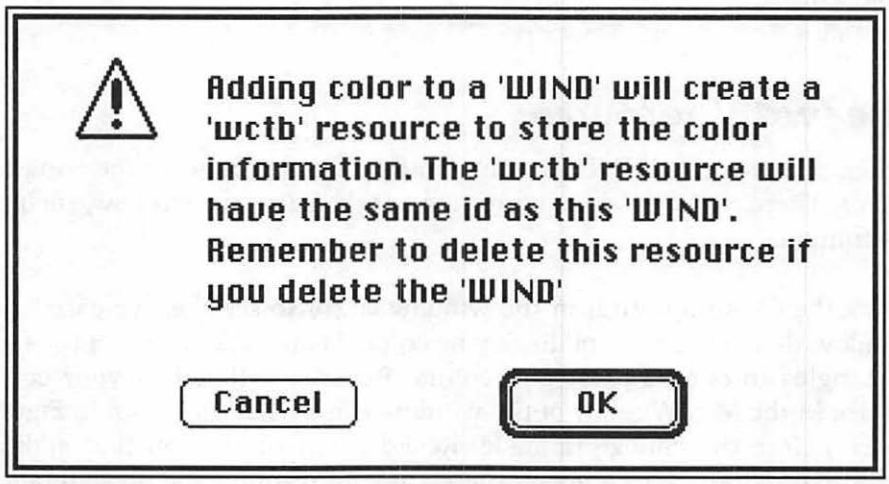


Figure 4–16. *ResEdit gives notification that a new resource will be added*

When you call *GetNewCWindow()* and pass the resource ID of a color 'WIND', the colors you added in ResEdit will be displayed on the window opening on the screen.

The Cursor

At the start of this chapter I packaged all of the Toolbox initialization calls into one function called *Initialize_Toolbox()*. The last call in that function is a call to *InitCursor()*, which sets the cursor to the familiar arrow shape. You've noticed in many Macintosh programs that the cursor can take on different shapes. Often it looks like an arrow, but it can also take on other forms. A word processor, for example, sets the cursor to an I-beam shape when it's over a window that allows editing.

As your program runs, you may want to change the appearance of the cursor. You can do that by using two Toolbox calls: *GetCursor()* and *SetCursor()*. The system defines five cursors for your use, and they're stored as resources in the system resource file.

InitCursor() sets the cursor to the default cursor, the arrow. For any of the other four cursors, use *GetCursor()* to get a handle to the desired one. You supply the resource ID of the 'CURS' resource you want to display. You don't have to know the 'CURS' IDs—the four system cursor resources are defined by constants: *iBeamCursor*, *crossCursor*, *plusCursor*, and *watchCursor*.

On a Macintosh handles can be of the generic *Handle* type or a type specific to the object being worked with. In the previous chapter you saw that a call to *GetPicture()* returns a *PicHandle*. A call to *GetCursor()* returns a *CurHandle* to your program. After getting a *CurHandle* to a cursor, call *SetCursor()* to actually make the cursor change shape. When passing the cursor handle to *SetCursor()*, dereference it once. *SetCursor()* is expecting a pointer to a cursor, and you've got a handle to one.

Here's an example that lets the user know a short wait is in order. It sets the cursor to the watch, does some task that takes some time, then sets the cursor back to the arrow.

```
CursHandle  watch_handle;

watch_handle = GetCursor( watchCursor );

HLock( (Handle)watch_handle );
   SetCursor( *watch_handle );
HUnlock( (Handle)watch_handle );
```

```
[ do some time-consuming stuff ]

InitCursor();
```

Yes, we did slip something new into the last code fragment. We sandwiched the call to *SetCursor()* between calls to *HLock()* and *HUnlock()*. The Toolbox routine *HLock()* marks a relocatable block as nonrelocatable. *HUnlock()* sets the block back to its normal condition of relocatable.

What makes this situation so unique that we needed to include this pair of calls, when we've never done so in the past? This is the first time we passed a dereferenced handle to a Toolbox routine. In Chapter 2 we discussed memory compaction. Memory compaction can take place during the execution of some Toolbox routines. If it does, and that routine is working with a dereferenced handle, the results can be unpredictable.

Remember, a handle holds the address of a master pointer. The master pointer won't ever move, but what it points to may. In our call to *SetCursor()* we're passing an address—the address held in the master pointer. Imagine that memory compaction takes place in the middle of the call to *SetCursor()*. We passed *SetCursor()* the address of the object, the cursor. If the block that this address points to moves, *SetCursor()* will not find the cursor. And that's a big problem.

We've recreated the above scenario in Figure 4–17. The handle holds the address of a master pointer—65000. Dereferencing a handle one time yields the contents of what it points to—the contents of the master pointer, or 80000. So that's what is being passed when **the_handle* is used as a parameter—80000. Just to complete our dereferencing story, if we dereferenced a second time we'd have the contents of address 80000, the object itself—the cursor. Now, what happens when we pass **the_handle*, or the address 80000 to *SetCursor()*, and memory gets compacted in the middle of the call? When *SetCursor()* looks for the cursor that should be at address 80000, it just might not be there!

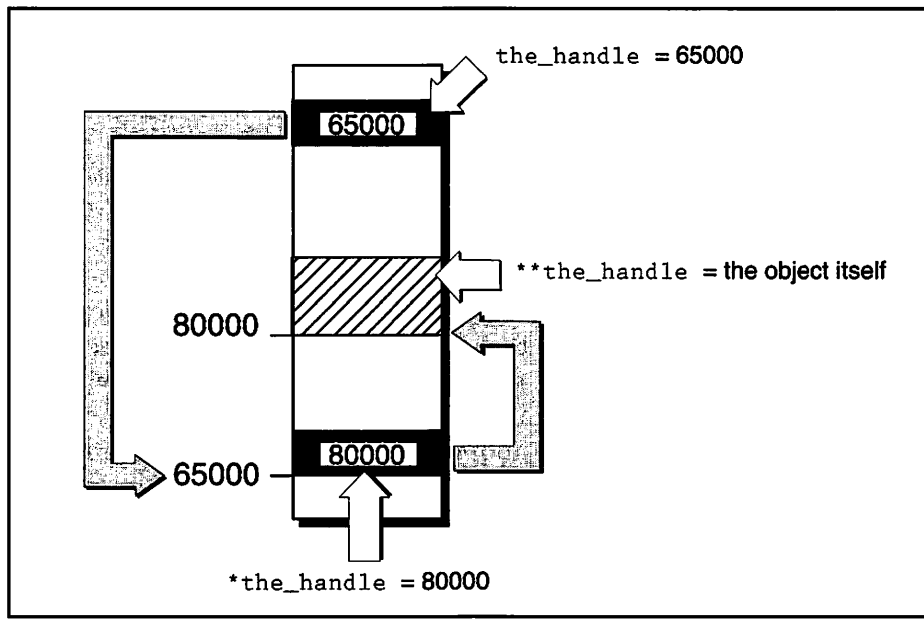


Figure 4–17. *Dangers of passing a dereferenced handle*

Nesting code between calls to *HLock()* and *HUnlock()* prevents the above situation from occurring. The relocatable block used as the parameter to *HLock()* will not move—even if the heap gets compacted. The advantage to this technique should be apparent—the Toolbox calls working with dereferenced handles will work successfully. The downside is that while a relocatable block is locked, it can cause memory fragmentation. That's why we unlock it immediately after our use of the dereferenced handle is complete.

Memory compaction takes place only at select times. Not all Toolbox calls are affected. The Inside Macintosh series lists the ones that may involve compaction. If you don't have this list, feel free to play it safe and call *HLock()* every time you pass an dereferenced handle. As long as you make sure to call *HUnlock()* when the call is complete, you can't go wrong.

Chapter Program: Drawing on the Mac

QuickDrawing, this chapter's example program, uses the same format as the preceding example in that it simply puts a single window on the screen and then does its stuff. In this case, the "stuff" is drawing.

If you run *QuickDrawing* on a color system it will open a window with a colored background and draw three colored shapes in the window. If you're running a monochrome system it will open a standard window and draw the shapes in black and white. Figure 4–18 shows *QuickDrawing's* window on a monochrome system. To quit the program, click the mouse.

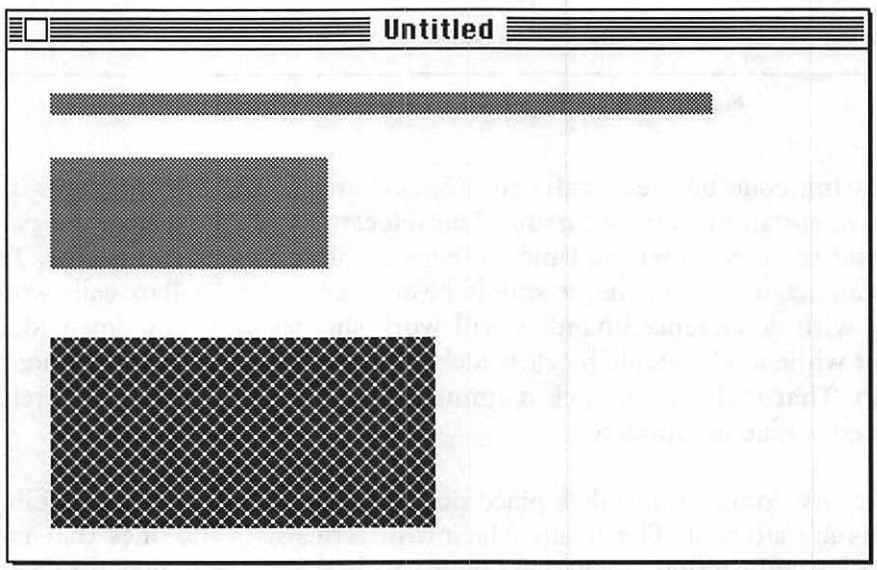


Figure 4–18. *Monochrome version of QuickDrawing's window*

Program resources: *QuickDrawing.π.rsrc*

QuickDrawing uses the four resource types discussed in this chapter. Figure 4–19 shows the program's two black and white 'PAT ' patterns and its two color 'ppat' patterns. These are the same as those used in earlier examples.

QuickDrawing has two 'WIND' resources. One is the standard window seen in the preceding chapters but enlarged just a little. The second 'WIND' is a color window. When you clicked the Custom radio button in the window editor ResEdit created a 'wctb' resource to hold the window's color information.

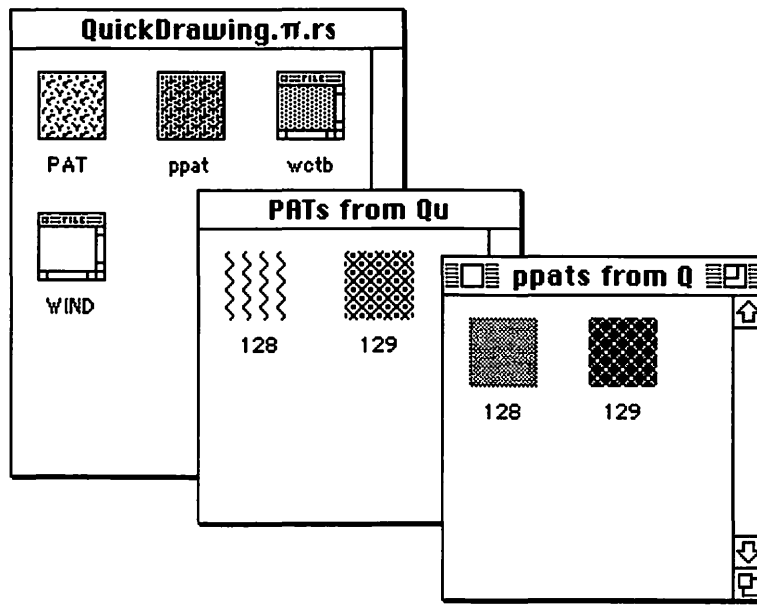


Figure 4–19. QuickDrawing's *resources*

Program listing: *ResourceUser.c*

Here is the source code in full.

```
/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++*/

#include <GestaltEqu.h>


/*+++++++++++++++++   Function prototypes   +++++++++++++++++*/

void  Initialize_Toolbox( void );
void  Open_Window( void );
```

```
void  Do_Drawing( void );
void  Draw_Color_Pen_Pattern( void );
void  Draw_Color_Fill_Pattern( void );
void  Draw_BW_Pen_Pattern( void );
void  Draw_BW_Fill_Pattern( void );


/*+++++++++++++   Define global constants   ++++++++++++++++++*/

#define    BW_WIND_ID                128
#define    C_WIND_ID                 129
#define    NIL                        0L
#define    IN_FRONT       (WindowPtr)-1L
#define    REMOVE_EVENTS              0

#define    PEN_PAT_BW_ID             128
#define    FILL_PAT_BW_ID            129
#define    PEN_PAT_COLOR_ID          128
#define    FILL_PAT_COLOR_ID         129


/*+++++++++++++++   Define global variables   +++++++++++++++++*/

WindowPtr    The_Window;
Boolean      All_Done = FALSE;
EventRecord  The_Event;
Boolean      Color_QD_Present;


/*++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   Initialize_Toolbox();

   Open_Window();

   Do_Drawing();

   while ( All_Done == FALSE )
   {
      GetNextEvent( everyEvent, &The_Event );

      switch ( The_Event.what )
      {
```

```
            case mouseDown:
               All_Done = TRUE;
               break;
         }
      }
}


/*++++++++++++++++++  Initialize the Toolbox  +++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*+++++++++  Open a single mono or color window   ++++++++++++*/

void  Open_Window( void )
{
   OSErr  err;
   long   response;

   err = Gestalt( gestaltQuickdrawVersion, &response );

   if ( ( err == noErr ) && ( response == gestaltOriginalQD ) )
      Color_QD_Present = FALSE;
   else
      Color_QD_Present = TRUE;

   if ( Color_QD_Present == TRUE )
      The_Window = GetNewCWindow( C_WIND_ID, NIL, IN_FRONT );
   else
      The_Window = GetNewWindow( BW_WIND_ID, NIL, IN_FRONT );

   if ( The_Window == NIL )
      ExitToShell();
}
```

```
/*+++++++  Branch to mono or color drawing routines   +++++++*/

void  Do_Drawing( void )
{
   GrafPtr       save_port;
   PenState      save_state;

   GetPort( &save_port );
   SetPort( The_Window );
   GetPenState( &save_state );

   if ( Color_QD_Present == TRUE )
   {
      Draw_Color_Pen_Pattern();
      Draw_Color_Fill_Pattern();
   }
   else
   {
      Draw_BW_Pen_Pattern();
      Draw_BW_Fill_Pattern();
   }

   SetPenState( &save_state );
   SetPort( save_port );
}


/*++++++++++++  Change pen pattern and draw in mono  +++++++++*/

void  Draw_BW_Pen_Pattern( void )
{
   PatHandle  pen_pat_handle;
   Rect       the_rect;

   pen_pat_handle = GetPattern( PEN_PAT_BW_ID );
   PenPat(**pen_pat_handle);

   PenSize( 10, 10 );
   MoveTo( 20, 20 );
   Line( 300, 0 );

   SetRect( &the_rect, 20, 50, 150, 100 );
   PaintRect( &the_rect );
```

```
}


/*++++++++++  Change pen pattern and draw in color   ++++++++*/

void  Draw_Color_Pen_Pattern( void )
{
   PixPatHandle  pen_pixpat_handle;
   Rect          the_rect;

   pen_pixpat_handle = GetPixPat( PEN_PAT_COLOR_ID );
   PenPixPat( pen_pixpat_handle );

   PenSize( 10, 10 );
   MoveTo( 20, 20 );
   Line( 300, 0 );

   SetRect( &the_rect, 20, 50, 150, 100 );
   PaintRect( &the_rect );
}


/*++++++++++++  Fill a shape using a mono pattern   ++++++++++*/

void  Draw_BW_Fill_Pattern( void)
{
   PatHandle  fill_pat_handle;
   Pattern    the_pattern;
   Rect       the_rect;

   fill_pat_handle = GetPattern( FILL_PAT_BW_ID );

   SetRect( &the_rect, 20, 150, 200, 250 );
   FillRect( &the_rect, **fill_pat_handle );
}


/*++++++++++  Fill a shape using a color pattern   ++++++++++*/

void  Draw_Color_Fill_Pattern( void)
{
   PixPatHandle  fill_ppat_handle;
   Rect          the_rect;

   fill_ppat_handle = GetPixPat( FILL_PAT_COLOR_ID );
```

```
    SetRect( &the_rect, 20, 150, 200, 250 );
    FillCRect( &the_rect, fill_ppat_handle );
}
```

Stepping through the code

Stepping through *QuickDrawing* will be a breeze. All of its code was developed in this chapter, and there are no surprises.

The #include directives

QuickDrawing checks the system for Color QuickDraw using *Gestalt()*, so you need to include the *GestaltEqu.h* header file.

```
#include <GestaltEqu.h>
```

The #define directives

You should now be aware that a good Mac program meets the user's needs, whatever they may be. *QuickDrawing*, to meet these needs, uses two 'WIND' resources to display either a monochrome window (*BW_WIND_ID*) or a color window (*C_WIND_ID*). *NIL* and *IN_FRONT* are used in *GetNewWindow()*, and *REMOVE_EVENTS* is used at initialization.

QuickDrawing will load either two monochrome 'PAT ' resources into memory, *PEN_PAT_BW_ID* and *FILL_PAT_BW_ID*, or two color 'ppat' resources: *PEN_PAT_COLOR_ID* and *FILL_PAT_COLOR_ID*.

```
#define    BW_WIND_ID                128
#define    C_WIND_ID                 129
#define    NIL                        0L
#define    IN_FRONT        (WindowPtr)-1L
#define    REMOVE_EVENTS              0

#define    PEN_PAT_BW_ID             128
#define    FILL_PAT_BW_ID            129
#define    PEN_PAT_COLOR_ID          128
#define    FILL_PAT_COLOR_ID         129
```

The main() function

QuickDrawing first performs initializations, as always. The only difference here is that the calls are bundled into a routine called *Initialize_Toolbox()*.

The program next opens a single window. Whether it's a color window or not is determined by *Open_Window()*. After that, *Do_Drawing()* sets up the drawing that will take place.

```
void  main( void )
{
   Initialize_Toolbox();

   Open_Window();

   Do_Drawing();

   while ( All_Done == FALSE )
   {
      GetNextEvent( everyEvent, &The_Event );

      switch ( The_Event.what )
      {
         case mouseDown:
            All_Done = TRUE;
            break;
      }
   }
}
```

Initialization

Here's *Initialize_Toolbox()*, exactly as put together earlier in this chapter.

```
void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
```

```
    FlushEvents( everyEvent, REMOVE_EVENTS );
    InitCursor();
}
```

Checking for color

Open_Window() performs the very necessary check for color using the *Gestalt()* function. For now, use it "as is." In Chapter 8 you'll see how to change the first parameter to check for all sorts of things, not just color. Once *GetNewWindow()* or *GetNewCWindow()* has been called, the program verifies that the window was indeed loaded into memory. If not, *The_Window* will be nil and the program ends.

```
void  Open_Window( void )
{
    OSErr  err;
    long   response;

    err = Gestalt( gestaltQuickdrawVersion, &response );

    if ( ( err == noErr ) && ( response == gestaltOriginalQD ) )
        Color_QD_Present = FALSE;
    else
        Color_QD_Present = TRUE;

    if ( Color_QD_Present == TRUE )
        The_Window = GetNewCWindow( C_WIND_ID, NIL, IN_FRONT );
    else
        The_Window = GetNewWindow( BW_WIND_ID, NIL, IN_FRONT );

    if ( The_Window == NIL )
        ExitToShell();
}
```

Preserving the environment

Do_Drawing() serves as a branching station for the drawing that's about to take place. One set of routines executes if Color QuickDraw is present, another set if it's not. In addition, *Do_Drawing()* guarantees that when the function is finished the state of the pen and the port will be returned to their previous conditions.

```
void  Do_Drawing( void )
{
   GrafPtr        save_port;
   PenState       save_state;

   GetPort( &save_port );
   SetPort( The_Window );
   GetPenState( &save_state );

   if ( Color_QD_Present == TRUE )
   {
      Draw_Color_Pen_Pattern();
      Draw_Color_Fill_Pattern();
   }
   else
   {
      Draw_BW_Pen_Pattern();
      Draw_BW_Fill_Pattern();
   }

   SetPenState( &save_state );
   SetPort( save_port );
}
```

Lines and shape painting

Both *Draw_BW_Pen_Pattern()* and *Draw_Color_Pen_Pattern()* do the same thing; that is, they change the pattern of the pen and then draw a line and paint a rectangle. The only difference is in the resource each uses to obtain the pattern and the call that is made to set the pen pattern.

```
void  Draw_BW_Pen_Pattern( void )
{
   PatHandle  pen_pat_handle;
   Rect       the_rect;

   pen_pat_handle = GetPattern( PEN_PAT_BW_ID );
   PenPat(**pen_pat_handle);

   PenSize( 10, 10 );
   MoveTo( 20, 20 );
   Line( 300, 0 );
```

```
    SetRect( &the_rect, 20, 50, 150, 100 );
    PaintRect( &the_rect );
}


void  Draw_Color_Pen_Pattern( void )
{
    PixPatHandle  pen_pixpat_handle;
    Rect          the_rect;

    pen_pixpat_handle = GetPixPat( PEN_PAT_COLOR_ID );
    PenPixPat( pen_pixpat_handle );

    PenSize( 10, 10 );
    MoveTo( 20, 20 );
    Line( 300, 0 );

    SetRect( &the_rect, 20, 50, 150, 100 );
    PaintRect( &the_rect );
}
```

Shape filling

Draw_BW_Fill_Pattern() and *Draw_Color_Fill_Pattern()* each draw an identically sized and placed rectangle. The BW routine uses a 'PAT ' resource, while the Color routine uses a 'ppat'.

```
void  Draw_BW_Fill_Pattern( void)
{
    PatHandle  fill_pat_handle;
    Pattern    the_pattern;
    Rect       the_rect;

    fill_pat_handle = GetPattern( FILL_PAT_BW_ID );

    SetRect( &the_rect, 20, 150, 200, 250 );
    FillRect( &the_rect, **fill_pat_handle );
}


void  Draw_Color_Fill_Pattern( void)
{
    PixPatHandle  fill_ppat_handle;
    Rect          the_rect;
```

```
    fill_ppat_handle = GetPixPat( FILL_PAT_COLOR_ID );

    SetRect( &the_rect, 20, 150, 200, 250 );
    FillCRect( &the_rect, fill_ppat_handle );
}
```

Chapter Summary

QuickDraw is a group of Toolbox routines and is the single largest group of Toolbox functions. Besides drawing the shapes and pictures you see displayed in windows, QuickDraw draws the window itself. In fact, QuickDraw is responsible for drawing everything on the Macintosh screen. QuickDraw, and other parts of the Toolbox, have to be initialized before use.

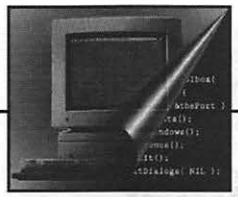
Every window has its own graphics port or environment. A graphics port defines what lines and text will look like. When you give each window its own graphics port, you allow different windows to display different styles of text and draw shapes of different patterns. You can change a graphics feature within a port by making a change to the port's graphics pen. The pen is invisible—it exists as a reference that aids you in manipulating graphics features.

You use Toolbox routines to tell QuickDraw what to draw. Because each window has its own graphics port, you must make sure that QuickDraw knows which window it should draw to in response to the commands you give it. Before you draw to a window, you'll give QuickDraw this information in the form of a call to *SetPort()*.

The primary shape that QuickDraw works with is the Rect, the C data type that represents a rectangle. By defining the boundaries of a rectangle, you give QuickDraw the information it needs to draw rectangles, ovals, and round rectangles (rectangles with rounded corners). The Toolbox contains a host of shape-drawing routines that allow you to frame, fill, invert, and erase these different types of shapes.

You can add flair to your shapes by using patterns. The C data type Pattern allows you to choose from several defined patterns. You can also define your own monochrome patterns using 'PAT ' resources.

Many Macintosh users now have color systems, and you can support these users by using Color QuickDraw. The color version of QuickDraw allows you to draw shapes in color, create color patterns using the 'ppat' resource type, and add color to the frame or content of windows.



5 Working with Windows

Windows are what originally set the Macintosh apart from other computers. To display information, a Macintosh needs either a window or the brother of the window, a dialog box. In this chapter you'll learn about window-handling techniques.

Let's begin with a discussion of events. Nothing happens to or with a window until an event occurs. A click of the mouse button is usually what a window responds to, so the focus will be on events involving the mouse.

Devising a system to handle events that involve one window is relatively straightforward. However, when more than a single window is on the screen, window-handling techniques become more complex. This chapter provides a strong background on the basic techniques of working with a window. It also covers the more difficult topic of working with multiple windows.

As do the previous chapters, this chapter finishes with a sample program that demonstrates the techniques highlighted in the chapter.

Windows Primer

Here is a concise summary of just what a window is.

The 'WIND' resource

A window starts as a 'WIND' resource, created here in ResEdit. Chapter 1 covered the 'WIND', so this chapter will simply show the 'WIND' editing window, shown here in Figure 5–1.

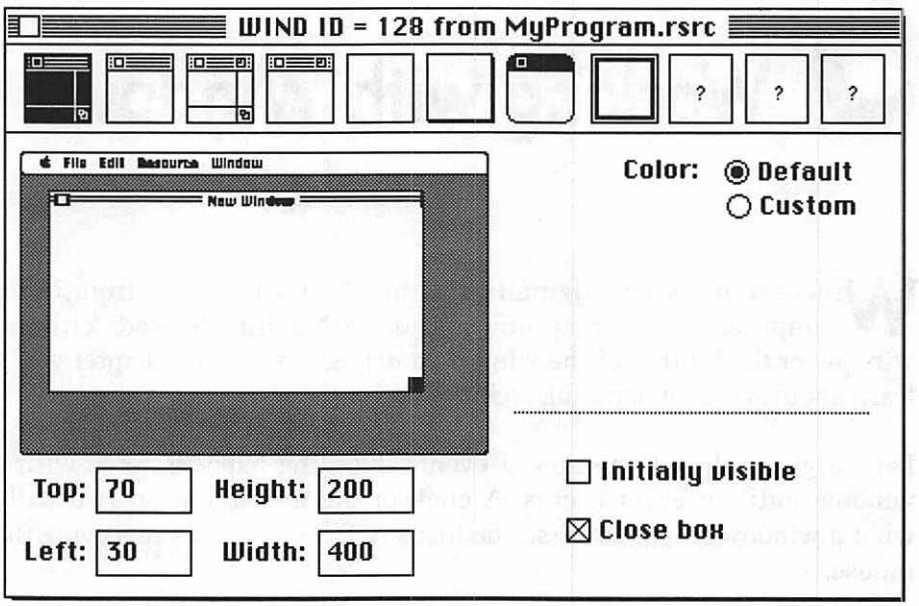


Figure 5–1. *A 'WIND' resource viewed in ResEdit*

Loading a 'WIND'

You've already seen the Toolbox routine *GetNewWindow()* in action several times. It loads a window into memory and returns a pointer to the window. Below is a call to *GetNewWindow()*.

```
#define     NIL                     0L
#define     IN_FRONT    (WindowPtr)-1
#define     WIND_ID                 128
```

```
WindowPtr  the_window;

the_window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
```

The first parameter passed to *GetNewWindow()* is the resource ID of the 'WIND' resource to use. The second parameter is a pointer that tells the Memory Manager where in memory to place the window. A value of 0L—the number zero followed by the letter L—is the convention used by many Macintosh programmers to serve as the value for a nil pointer. It tells the Window Manager to allocate the memory for you. (Later in this chapter you'll use a value other than nil for the storage.) The third parameter signals the Window Manager to place the new window behind all others (0) or in front of all others (-1). The Toolbox is looking for a *WindowPtr* here, so you'll have to *cast* the value (as shown by *WindowPtr* in parentheses) so that the compiler does not produce an error or warning message.

The *WindowRecord*, *WindowPtr* and *WindowPeek*

Every window is, in a sense, a world unto itself. Each window can have its individual properties, such as the size and font of the text it will display and whether the window is visible at this moment. The data structure *WindowRecord* holds this information. Here's the structure:

```
struct WindowRecord
{
   GrafPort       port;
   short          windowKind;
   Boolean        visible;
   Boolean        hilited;
   Boolean        goAwayFlag;
   Boolean        spareFlag;
   RgnHandle      strucRgn;
   RgnHandle      contRgn;
   RgnHandle      updateRgn;
   Handle         windowDefProc;
   Handle         dataHandle;
   StringHandle   titleHandle;
   short          titleWidth;
```

```
    ControlHandle   controlList;
    struct          WindowRecord *nextWindow;
    PicHandle       windowPic;
    long            refCon;
};
```

The heart of the *WindowRecord* is the very first member, the port member: *GrafPort*. Recall from Chapter 4 that a *GrafPort* holds all the information about a graphics port, which is a drawing environment.

You won't need to memorize the exact makeup of the *WindowRecord* structure. Instead, you'll work with *WindowPtrs*. A *WindowPtr* points to the *GrafPort* of a *WindowRecord*. Once you have a *WindowPtr* you can do just about anything you want to a window through Toolbox calls. You call the Toolbox routine name and include the pointer to the window you want to work with as follows:

```
#define     NIL                     0L
#define     IN_FRONT    (WindowPtr)-1
#define     WIND_ID                 128

WindowPtr   the_window;

the_window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

SetPort( the_window );     /* set the port to the new window */

ShowWindow( the_window ); /* show the window on the screen  */
```

In addition to a *WindowPtr* there is also a Macintosh C type called a *WindowPeek*. *WindowPtr* points specifically to the *GrafPort* of the *WindowRecord*, but *WindowPeek* points to the entire *WindowRecord* structure. Figure 5–2 illustrates this.

From Figure 5–2 it appears that the *WindowPeek* is more powerful, since it allows access to all of the members of a *WindowRecord*, not just the port. That is, in fact, true. But there are many instances where you won't need to access any of the members other than the port. In those cases it's fine to use the *WindowPtr*. You'll also use a *WindowPtr* because many Toolbox routines expect a *WindowPtr* as one of the parameters, and do not expect a *WindowPeek*.

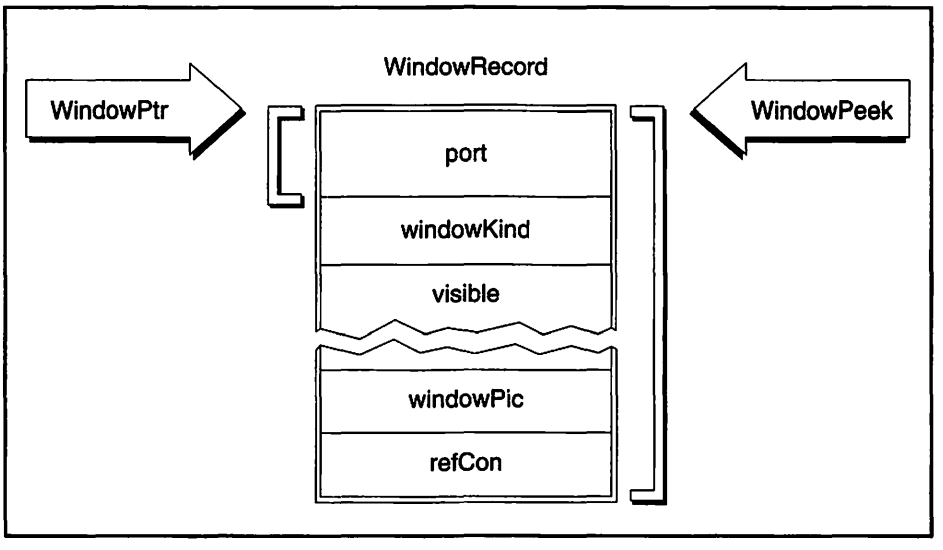


Figure 5-2. *A* WindowPtr *and* WindowPeek

Just when should you use a *WindowPeek*? You'll find out in the highlight of this chapter, the section that deals with working with multiple windows.

Event Handling

Chapter 1 pointed out that the event loop is a distinguishing feature of programs written for the Macintosh and other GUI systems. A Macintosh program calls *GetNextEvent()* to retrieve an event, then processes that event. How it does that is dependent on the type of event retrieved. Below is the *main()* function for a typical Mac program.

```
#include <Traps.h>

Boolean  All_Done = FALSE;
Boolean  Multifinder_Present;

void  main( void )
{
   Initialize_Toolbox();
```

```
    Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                           NGetTrapAddress(_Unimplemented, ToolTrap));

    while ( All_Done == FALSE )
       Handle_One_Event();
}
```

The last chapters covered the *Initialize_Toolbox()* routine. There, *Initialize_Toolbox()* called the eight Toolbox initialization routines that you should call at the start of every program.

The next line determines whether the system on which the program is running has Apple's MultiFinder. It then sets a *Boolean* variable based on this information. MultiFinder allows the running of multiple applications and is built into System 7. Before System 7, however, it was a separate, optional program.

From here on you're going to see this MultiFinder test in every program. At this point, I recommend that you accept at face value this rather confusing-looking line of code. Chapter 8 fully describes what's going on. Take note that the header file *Traps.h* is included in the listing—it's required so that the compiler understands constants such as *_WaitNextEvent*. All the remaining programs at the end of each chapter include this header.

The last two lines of *main()* are repeated until the program terminates. The *while* statement contains a check to see if the *All_Done* variable is still true. If it is true, the program retrieves and handles an event.

The general approach to handling a single event is to determine the type of the event, then branch to a routine that handles that particular event type. First, determine the general type of event, such as the updating of a window or a click of the mouse button. For a click of the mouse button, further determine the location of the cursor when the mouse button was pressed. Figure 5–3 shows this branching technique.

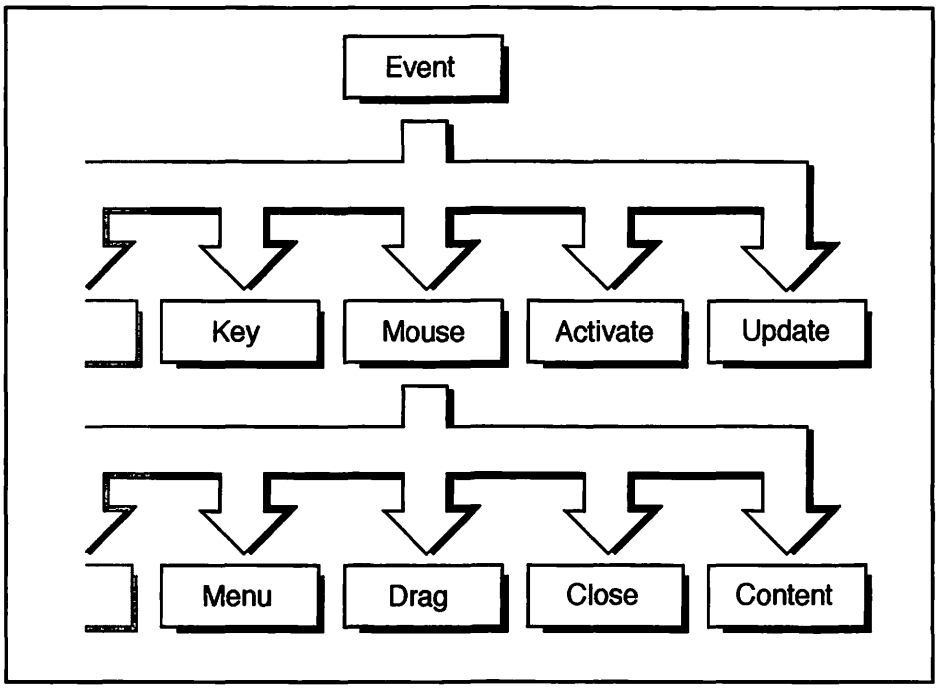


Figure 5–3. *An event leads to branching*

Figure 5–3 shows only a few of the event types, you can assume that there are several more types off to the left. The same is true of the location of a mouse event. Keeping Figure 5–3 in mind, take a look at what a *Handle_One_Event()* routine might look like.

```
void  Handle_One_Event( void )
{
   if ( Multifinder_Present == TRUE )
      WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION);
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
      case keyDown:
         Handle_Key_Down();
```

```
            break;

         case mouseDown:
            Handle_Mouse_Down();
            break;

         case activateEvt:
            Handle_Activate();
            break;

         case updateEvt:
            Handle_Update();
            break;
      }
   }
```

The *Handle_One_Event()* routine first uses the *Multifinder_Present* variable to determine which of two event-retrieving routines the program should use: *GetNextEvent()* or *WaitNextEvent()*. The difference between the two? *WaitNextEvent()* will relinquish the Macintosh CPU to other applications periodically. This has the effect of allowing other applications to work in the background. *GetNextEvent()* is an older routine and isn't supportive of multitasking in the same way.

The obvious question is: why, then, would you ever use *GetNextEvent()*? Because *WaitNextEvent()* is only available on more recent machines. This explains the reasoning for performing the check in *main()* that sets the *Multifinder_Present* variable.

As mentioned, a complete discussion of the *Multifinder_Present* check, *GetNextEvent()*, and *WaitNextEvent()* appears in Chapter 8. You may want to skip the rather intense details behind all this until later. The remainder of this chapter will not be dependent on what is discussed in Chapter 8.

If the program calls *GetNextEvent()* it also calls *SystemTask()*. A call to this routine gives any open desk accessories a slice of processor time and allows it to do necessary housekeeping. If the program calls *WaitNextEvent()*, a call to *SystemTask()* is not necessary—the functions of *SystemTask()* are built right into *WaitNextEvent()*.

Housekeeping? As an example, open the Alarm Clock from the Apple menu, then run an application of your choice. Note that though the Alarm Clock window is in the background, it still ticks off seconds. Processor time is shared by the running application and the Alarm Clock desk accessory.

After retrieving an event, th *Handle_One_Event()* routine uses a *switch* statement to branch off to a utine written to handle just one type of event. In this example I call event types *keyDown* and *mouseDown*, among others. These are Apple-defined values, and there are a few more than appear in the example. Here's the complete list:

```
everyEvent  = -1
nullEvent   =  0
mouseDown   =  1
mouseUp     =  2
keyDown     =  3
keyUp       =  4
autoKey     =  5
updateEvt   =  6
diskEvt     =  7
activateEvt =  8
osEvt       = 15
```

You write the routines to handle an event. Some will be relatively short and straightforward. The handling of a mouse click, on the other hand, is more involved. That's because a mouse click can occur on different objects on the screen. Below is an example of the handling of a mouse click.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
```

```
            /* Handle click in the menu bar */
            break;

        case inSysWindow:
            /* Handle click in a desk accessory */
            break;

        case inDrag:
            /* Handle click in a window drag bar */
            break;

        case inGoAway:
            /* Handle click in a window close box */
            break;

        case inContent:
            /* Handle click in the content region of a window */
            break;
    }
}
```

The first thing *Handle_Mouse_Down()* does is to call *FindWindow()*. This Toolbox routine determines where the cursor is on the screen when the mouse button is pressed. If it is over a window, *FindWindow()* will return a pointer to that particular window.

Handle_Mouse_Down() then handles the event depending on the screen location, (or part of the screen) where the cursor is located. The routine uses a *switch* statement to reach the code used to handle a mouse click on a specific screen part. Here I've elected to show comments rather than the source code, I'll cover the code at appropriate places in this book. The *part codes*, such as *inMenuBar*, are Apple constants. Here's the entire list:

```
inDesk      = 0
inMenuBar   = 1
inSysWindow = 2
inContent   = 3
inDrag      = 4
inGrow      = 5
inGoAway    = 6
inZoomIn    = 7
inZoomOut   = 8
```

As you may have noticed, I love to reinforce a point with a figure. So to summarize event handling I give you Figure 5–4. As you study the figure, keep in mind that it shows only a few of the possible event types and only a few of the screen parts.

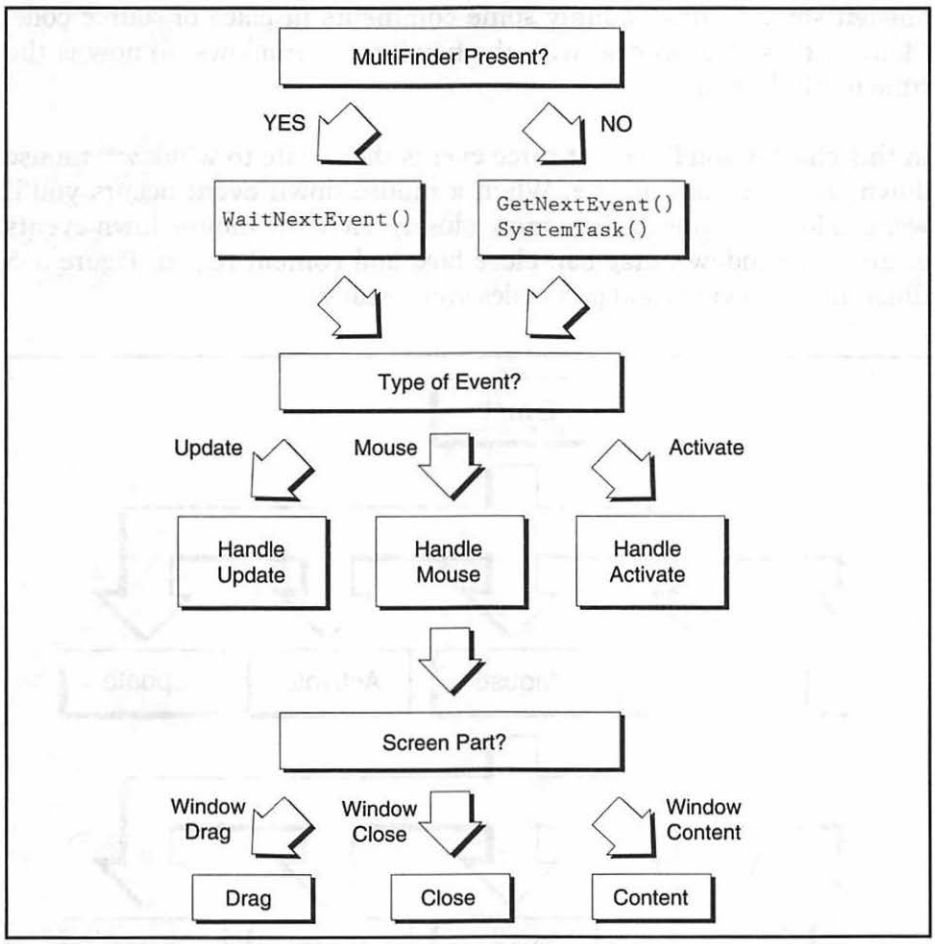


Figure 5–4. *Summary: handling of an event*

Lesson 5–1: Handling Events

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Windows and Events

In case you forgot, this chapter is about windows. The previous discussion of events was a prerequisite to any serious explanation of windows. The previous section outlined how the processing of an event takes place but left some blanks—mainly some comments in place of source code. Many of those blanks deal with the handling of windows, so now is the time to fill them in.

In this chapter you'll look at three events that relate to windows: mouse down, activate, and update. When a mouse down event occurs you'll want to look at things a little more closely. Here the mouse down events occur in a window's drag bar, close box, and content region. Figure 5–5 illustrates the events and part codes you'll study.

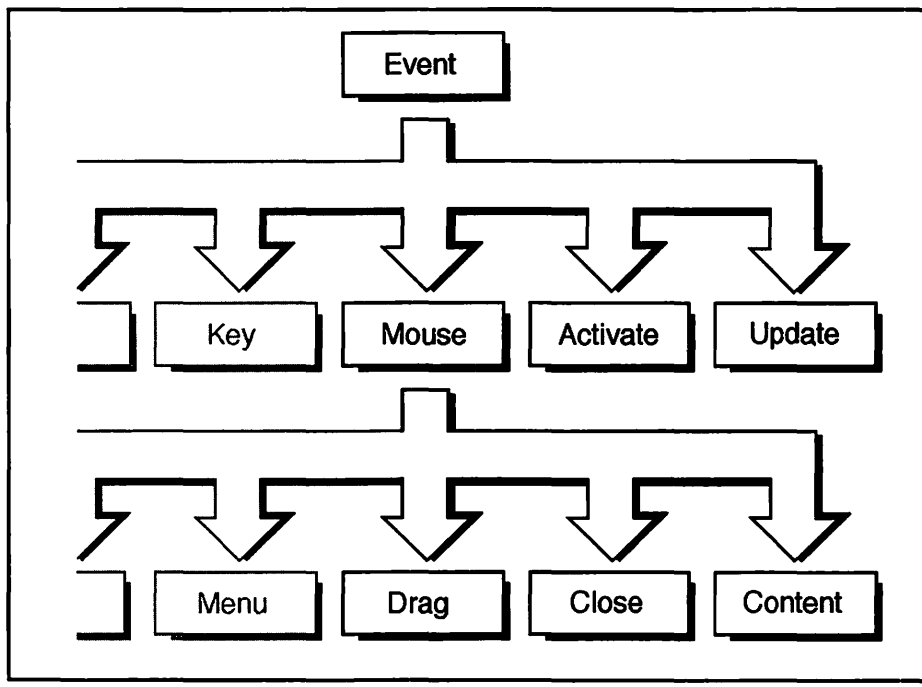


Figure 5–5. *Events and part codes relating to windows*

By covering the handling of these particular events and part codes you'll have a sound background for the finale of this chapter—the handling of multiple windows.

Mouse down events

When a click of the mouse button occurs in a window, you'll want to determine whether the click occurred in the window's drag bar, close box, or content region. You'll then react accordingly.

Handling a mouse click in a drag bar

Handling a mouse click in a window's drag bar is easy, thanks to the Toolbox routine *DragWindow()*. You need just one line in *Handle_Mouse_Down()*:

```
DragWindow( the_window, The_Event.where, &screenBits.bounds );
```

Here's that line in the context of *Handle_Mouse_Down()*:

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      [other code here]

      case inDrag:
         DragWindow(the_window, The_Event.where, &screenBits.bounds);
         break;

      [other code here]
   }
}
```

Once called, the *DragWindow()* routine takes control until the mouse button is released. While the user holds the mouse button down and moves the mouse, *DragWindow()* moves the window to follow the motion of the mouse.

The user can move a window about the screen by clicking the mouse button and holding it down while over the window's drag bar. To prevent

the user from dragging the window off the edge of a screen and entirely hiding it, you will create a boundary rectangle that defines the drag limits.

Macintosh provides you with a system global variable named *screenBits.bounds* that defines the pixel boundaries of the monitor your program is running on. This variable always holds the boundaries of the monitor's screen, regardless of the monitor's size.

I first declare a global *Rect* variable called *Drag_Rect* and then set it to the same size as *screenBits.bounds*. Next, I inset this rectangle a few pixels. The inset value represents the amount of a window, in pixels, that must always remain on the screen no matter how far off the edge of the screen the user drags a window. I've bundled this short bit of code into a routine called *Set_Window_Drag_Boundaries()*.

```
#define   DRAG_EDGE   10

Rect  Drag_Rect;

void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = screenBits.bounds;
   Drag_Rect.left += DRAG_EDGE;
   Drag_Rect.right -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}
```

This *screenBits.bounds* method assumes that your program is running on a system with only one monitor. Chapter 8 discusses a technique for establishing a boundary rectangle for dual-monitor systems.

Once created, you'll be able to use this drag boundary rectangle anytime, thus the reasoning for making *Drag_Rect* a global variable.

Handling a mouse click in a close box

Should the user depress the mouse button, the Toolbox routine *TrackGoAway()* then follows the movement of the mouse. If the user releases the button while over the close box of a window, the routine returns a value of true. A couple of simple housekeeping calls are all that's needed to then close the window. Here's a fragment that demonstrates *TrackGoAway()*:

```
if ( TrackGoAway( the_window, The_Event.where ) )
{
   HideWindow( the_window );
   DisposeWindow( the_window );
}
```

The call to *HideWindow()* is not necessary, but recommended. If a window has controls, such as scroll bars, then the housekeeping becomes more involved than shown here. You'll want the window hidden so that clean up goes on behind the scenes. *DisposeWindow()* closes a window and frees up the memory it uses.

Here's *TrackGoAway()* in the context you'll use it in your *Handle_Mouse_Down()* routine.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      [other code here]

      case inGoAway:
         if ( TrackGoAway( the_window, The_Event.where ) )
         {
            HideWindow( the_window );
            DisposeWindow( the_window );
         }
         break;
```

```
        [other code here]
    }
}
```

Handling a mouse click in a content region

If many cases, a mouse button click in the content area of a window requires that you simply make the window active, if it isn't already so. The *FrontWindow()* routine returns a pointer to the frontmost window. Compare this pointer to the pointer of the clicked window. If different, make a call to *SelectWindow()*. This routine takes care of selecting a window by providing the proper highlighting to the clicked-on window.

```
if ( the_window != FrontWindow() )
   SelectWindow( the_window );
else
{
   /* handle the needs, if any, of a click in */
   /* the contents of an active window        */
}
```

What if your program uses a window that does more than simply display information? Then you must write your program so that it is prepared to do more than just highlight a window. What else should it do? You'll see when you get to the end of the chapter. There I provide a concrete programming example along with the theory.

Here's the code as you'd see it within the *Handle_Mouse_Down()* routine.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      [other code here]

      case inContent:
```

```
            if ( the_window != FrontWindow() )
               SelectWindow( the_window );
            else
            {
               /* handle the needs, if any, of a click in */
               /* the contents of an active window        */
            }
            break;

         [other code here]
      }
   }
```

Handling mouse clicks in other areas

This version of *Handle_Mouse_Down()* also looks for a mouse click in the menu bar (*inMenuBar*) and in a desk accessory (*inSysWindow*). Chapter 7 covers a click in the menu bar.

A window that your application creates and manages is an application window. A desk accessory is a system window. As you've seen in the previous several sections, you are responsible for handling an event in an application window. The Macintosh is kind enough to handle an event in a desk accessory. One call to *SystemClick()* and you're all set.

SystemClick() takes control and processes the event by activating the desk accessory, dragging its window, and closing it, or whatever other action is appropriate. The code that makes up the desk accessory will then be responsible for handling subsequent events.

Now that you know how to handle each of the part codes you'll be using, it's time to look at the completed *Handle_Mouse_Down()* routine.

```
EventRecord  The_Event;

void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );
```

```
    switch ( the_part )
    {
       case inMenuBar:
          break;

       case inSysWindow:
          SystemClick( &The_Event, the_window );
          break;

       case inDrag:
          DragWindow(the_window, The_Event.where, &screenBits.bounds);
          break;

       case inGoAway:
          if ( TrackGoAway( the_window, The_Event.where ) )
          {
             HideWindow( the_window );
             DisposeWindow( the_window );
          }
          break;

       case inContent:
          if ( the_window != FrontWindow() )
             SelectWindow( the_window );
          else
          {
             /* handle the needs, if any, of a click in */
             /* the contents of an active window        */
          }
          break;
    }
}
```

I'll fill in the code for the *inContent* case in the Multiple Window Techniques section. This section also discusses dealing with activate and update events.

Single Window Techniques

The *Handle_One_Event()* routine is the hub from which your program branches off to handle a particular event. So far, the focus has been on a

mouse down event. For window handling you should be aware of two other event types: activates and updates.

Activate events

Macintosh programs have one and only one window active at any given time. The active window is the window that responds to user actions such as keystrokes or a click of the mouse. If there is more than one window on the screen, the active window is the frontmost of them. The drag bar of the active window has a highlighted appearance that sets it apart from other windows.

For a program with more than one window, a click on a deactivated window will generate two activate events, one to signify the deactivation of the frontmost window and one to signify the activation of the clicked-on window. The Window Manager handles the changing highlight conditions of window frames; you will be responsible for handling changes to the content of a window.

For a program that creates only one window, it is not uncommon to omit code that handles an activate event. That's because only one activate event will occur in a program of this type. When the window is first created, *GetNewWindow()* will generate an activate event.

I have more to say about activate events later in this chapter when I discuss the handling of multiple windows.

Updating a window

When a covered, or obscured, window becomes exposed, its contents will need updating; that is, you need to redraw what is in the window. A window that needs updating will trigger the occurrence of an update event. It becomes your job to handle the event. Begin by branching from *Handle_One_Event()* to a routine that handles an update—aptly named *Handle_Update()*.

```
void  Handle_One_Event( void )
{
```

```
    [other code here]

    switch ( The_Event.what )
    {
       case updateEvt:
          Handle_Update();
          break;

       [other code here]
    }
}
```

Here's a typical *Handle_Update()* routine that updates a window in a program that has a single application window.

```
void  Handle_Update( void )
{
   WindowPtr  the_window;
   GrafPtr    old_port;

   the_window = ( WindowPtr )The_Event.message;

   GetPort( &old_port );
   SetPort( the_window );

   BeginUpdate( the_window );
      EraseRgn( &the_window->visRgn );
      Draw_Something( the_window );
   EndUpdate( the_window );

   SetPort( old_port );
}
```

Because I like to use techniques that apply to all sorts of programs, I'll write *Handle_Update()* in such a manner that you can use it, with some modification, in a program that has more than one window.

Handle_Update() first uses the message element of *The_Event* to determine which window needs updating, in case there is more than one window on the screen. The Event Manager conveniently places a pointer to the window that needs updating in the message element. Next, call *GetPort()* and *SetPort()*. You encountered *GetPort()* and *SetPort()* in the previous chapter.

A common mistake in window updating is forgetting to set the port. If there is more than one window on the screen, QuickDraw will draw to the window whose port is current, regardless of whether that window needs the updating or not.

In between the port operations you update the window. An update involves these steps:

- A call to *BeginUpdate()*.
- A call to *EraseRgn()*, passing the window's visible region.
- Drawing of the window contents.
- A call to *EndUpdate()*.

You're at this point in your code because there's an update event in the event queue. The Mac knew a window had become exposed and placed the event there. What the Macintosh doesn't know on its own is when you handle the update. The calls to *BeginUpdate()* and *EndUpdate()* tell the Mac just that, and let the computer know it should remove the update event from the queue. Note the indented code between these two calls. This is this book's convention, intended to clarify the logic of this routine.

The Window Manager at all times keeps track of the portion of a window that is exposed, or visible. It keeps this area in the *visRgn* element of the window's *WindowPtr* structure. A call to *BeginUpdate()* causes the Window Manager to save this value, and then temporarily set the visible region to that area of the window that was obscured. When you draw the contents of the window, QuickDraw will be limited to drawing in only this temporarily visible region. The result is that QuickDraw doesn't update the entire window—only the part that was formerly obscured. Figure 5–6 illustrates this.

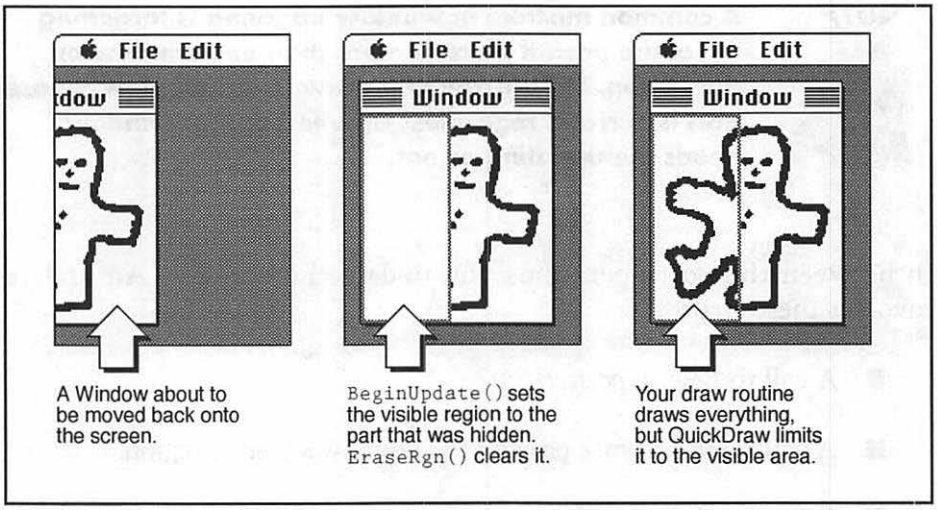


Figure 5–6. *Updating a window*

Now let's talk about the rather vaguely named routine *Draw_Something()*. What does this routine draw? The contents of the entire window. Why redraw everything when only a portion of it may need updating? Remember, the call to *BeginUpdate()* will tell QuickDraw what part of the window to draw to. When done, the call to *EndUpdate()* resets the window's visible region to its actual area, not to just the newly exposed area.

What your *Draw_Something()* routine will look like is entirely dependent on your application. In Figure 5–6 you can see that the content of the window is simply a picture—the display of a 'PICT' resource. Chapter 3 covered 'PICT' resources and displaying pictures in a window. For the window shown in Figure 5–6 the code for *Draw_Something()* would look like this:

```
#define   WILD_MAN_PICT_ID   128

void  Draw_Something( WindowPtr the_window )
{
   PicHandle  the_pict;
   Rect       pict_rect;
   short      pict_width, pict_height;
   GrafPtr    old_port;
```

```
    GetPort( &old_port );
     SetPort( the_window );

     the_pict = GetPicture( WILD_MAN_PICT_ID );

     pict_rect = ( **( the_pict ) ).picFrame;
     pict_width  = pict_rect.right - pict_rect.left;
     pict_height = pict_rect.bottom - pict_rect.top;
     SetRect ( &pict_rect, 10, 10, 10 + pict_width, 10 + pict_height );

     DrawPicture( the_pict, &pict_rect );

     SetPort( old_port );
}
```

As this example demonstrates, you update a window by actually going through all of the work of redrawing the contents of the window. If you haven't programmed a Mac in the past, you may have assumed you somehow get a "snapshot" of the contents of a window, then simply display that picture whenever appropriate.

Window updating is an important topic. An improper window update is immediately noticeable to the user in the form of a blank white area in a window or the appearance of graphics in the wrong part of the window (or even in the wrong window). For those reasons we'll take a look at another example.

Displaying a picture is easy; it's an operation that is unchanging. But what if some or all of a window's contents depend on information the user supplied? Consider this example: Your program asks the user to enter the four coordinates of a rectangle, then draws the rectangle. Later in the program, the user moves the window partially off screen, then back on. An update event is generated, and your *Draw_Something()* routine is called. Did you save those four values to some global variable, such as a *Rect*? Of course you did. If you hadn't, there would be no way to reproduce the rectangle now. Below is a code fragment to clarify this example.

```
Rect     Display_Rect;        /* global - hold the rectangle */
Boolean  Draw_Rect = FALSE;   /* global - used in updating   */

void  Get_Data_From_User( void )
{
   short  l, r, t, b;

   [ read in values l, r, t, and b here ]

   SetRect( Display_Rect, l, t, r, b );

   Draw_Rect = TRUE;
}


void  Draw_Something( WindowPtr the_window )
{
   [ other code here ]

   if ( Draw_Rect == TRUE )
      Frame_Rect( Draw_Rect );
}
```

From this example you can see that your updating routine might get quite involved and may contain decision-making logic, like the check of the *Draw_Rect* flag in the above example.

Simple window techniques

Before finishing this chapter with an example program that works with multiple windows, let's quickly cover some simple techniques that you can use in any program that has windows—one window or more. The following sections describe several simple window manipulations. All revolve around using the correct Toolbox call to perform the task at hand.

Lesson 5-2: Window Updating

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Moving a window

When you create a 'WIND' window resource in ResEdit you have the option of specifying whether a call to *GetNewWindow()* displays the window when it loads the 'WIND' into memory. It is best to mark the 'WIND' resource as invisible, as shown in Figure 5–7. Then, after you load the window you can, unbeknownst to the user, move the window to wherever you want on the screen and show it.

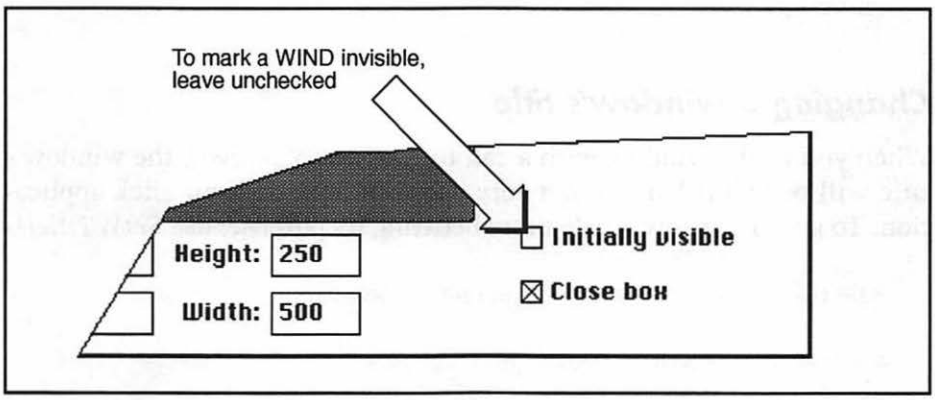


Figure 5–7. *Using ResEdit to mark a 'WIND' as invisible*

Knowing that you can take control from the user and move a window on your own should give you the idea that you can load a window, center it on the user's screen, and finally display it. You can, but your program has to have knowledge of the size of the user's screen—and that can vary from user to user. Centering a window on a Mac, regardless of monitor size, is just one of many topics that Chapter 8 covers.

Back to the topic: moving a window. Use the *MoveWindow()* routine, passing a pointer to the window you want to move, the pixel coordinates of the screen location to move the window to, and a *Boolean* value that tells whether to activate (highlight) the window.

```
#define    LEFT    20    /* 20 pixels from left of screen     */
#define    TOP     50    /* 50 pixels down from top of screen */

WindowPtr  the_window;
Boolean    activate_wind = TRUE;

MoveWindow( the_window, LEFT, TOP, activate_wind );
```

Showing and hiding a window

Earlier you learned that you can make a window invisible, or hidden, by using *HideWindow()*. You can make the same window visible again with a call to *ShowWindow()*. Here's an example:

```
WindowPtr  the_window;

HideWindow( the_window );
ShowWindow( the_window );
```

Changing a window's title

When you load a window with a call to *GetNewWindow()*, the window's title will be "Untitled"—not a very polished look for your slick application. To give a window a title more befitting its purpose, use *SetWTitle()*.

```
#define   NEW_WIND_TITLE   "\pGraphics Window"

WindowPtr  the_window;

SetWTitle( the_window, NEW_WIND_TITLE );
```

Multiple Window Techniques

A program that is capable of putting more than one window on the screen has a special set of needs that you must meet. There is a new twist to window updating: the contents of one window might not be the same as those of another window. This means that you don't have the luxury of simply calling on one generic update routine to handle any and all updates.

You'll need to devise a strategy that allows your program to distinguish between different types of windows. In this section you'll do just that. Imagine that you want to create a program that puts two types of windows on the screen. One window will be a control window with two buttons—one for drawing a shape, and one for erasing the shape. The second type of window will be a drawing window that displays the drawn shape. Additionally, the program will open more than one drawing window.

From the program description you may have surmised that there are a few extra challenges presented by a program capable of working with multiple

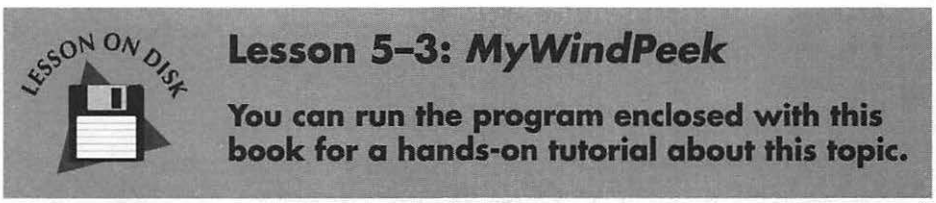
windows, challenges that you did not have to worry about when you planned out a program that would make use of just one window. Here they are:

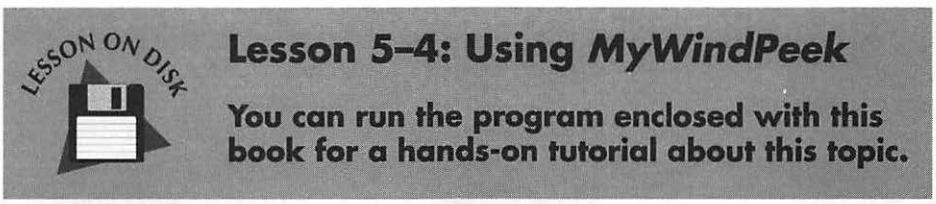
- An update event must be handled in two different ways, depending on which type of window needs updating.
- Once it has been determined that the update event corresponds to a drawing window, you must then determine which drawing window the event applies to.
- The user must be allowed to choose which of the drawing windows a click in the control window corresponds to.

These points make it clear that some planning is in order. Let's begin the plan by examining a method that allows the addition of window information to the window's existing *WindowRecord* structure.

Expanding the WindowRecord

You know from earlier in this chapter that a *WindowPtr* points to a window. More specifically, it points to the *port* member of a *WindowRecord* that holds the information about the window. You also know that you use a *WindowPeek* to gain access to the entire *WindowRecord*; not just the *port*. The following figure, Figure 5–8, appeared at the start of this chapter. I use it again to drive home the difference between a *WindowPtr* and a *WindowPeek*.





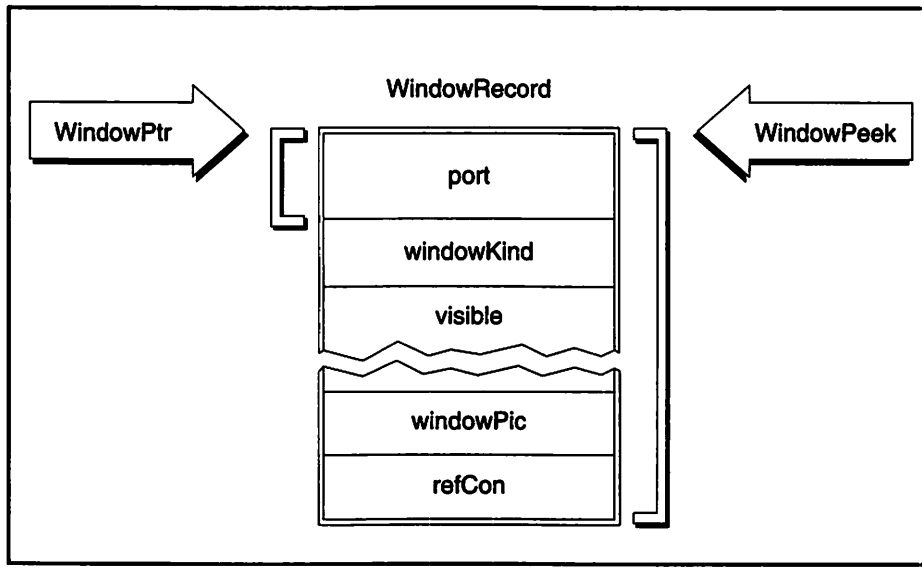


Figure 5–8. *A* WindowPtr *and* WindowPeek

When you call *GetNewWindow()* the Window Manager selects a chunk of memory and puts the window information—based on the 'WIND' resource—in that memory. Your program receives a pointer to the *GrafPort*. You can declare a *WindowPeek* and then use typecasting on the *WindowPtr* to access the window's entire *WindowRecord*. Here's an example.

```
#define     NIL                     0L
#define     IN_FRONT    (WindowPtr)-1
#define     WIND_ID                 128

WindowPtr   the_window;
WindowPeek  the_wind_peek;

the_window = GetNewWindow( WIND_ID, NIL, IN_FRONT );

the_wind_peek = ( WindowPeek )the_window;
```

This is the standard way to call *GetNewWindow()*, and to create a *WindowPeek* variable. There's another method you can use to create your own version of a WindowPeek that enables you to store and access extra information along with a *WindowRecord*.

This method involves creating your own data type by way of the C *typedef* keyword. Here's one example:

```
typedef  struct
{
   WindowRecord   wind_rec;
   short          wind_type;
   Boolean        drawn_in;

}  MyWindRecord, *MyWindPeek;
```

This definition creates a structure that has three members. The first member is a *WindowRecord*. The remaining two members give additional information about a window. Let's specify the type of the window in the *wind_type* member, and whether the window currently has a drawing in it with the *drawn_in* member. Like any structure, you can have as many or as few members as you want—whatever makes sense for your application.

The typedef names this new type *MyWindRecord*. It also creates a type that is a pointer to the structure—*MyWindPeek*. You know that a variable of the Macintosh C type *WindowPeek* points to an entire *WindowRecord*. What will a variable of your *MyWindPeek* point to? A *WindowRecord* and some extra information. Figure 5–9 illustrates the difference between a *WindowPeek* and your *MyWindPeek*.

Note in Figure 5–9 that both a *WindowPeek* and *MyWindPeek* begin by pointing to the start of a *WindowRecord*. What's at the start of a *WindowRecord*, and what is the very first member of the *WindowRecord*? The *port* member—the window's *GrafPort*. Making the first member of the data type a *WindowRecord* was not an accident. It allows you to use a variable of *MyWindPeek* anywhere that you would normally use a *WindowPeek*.

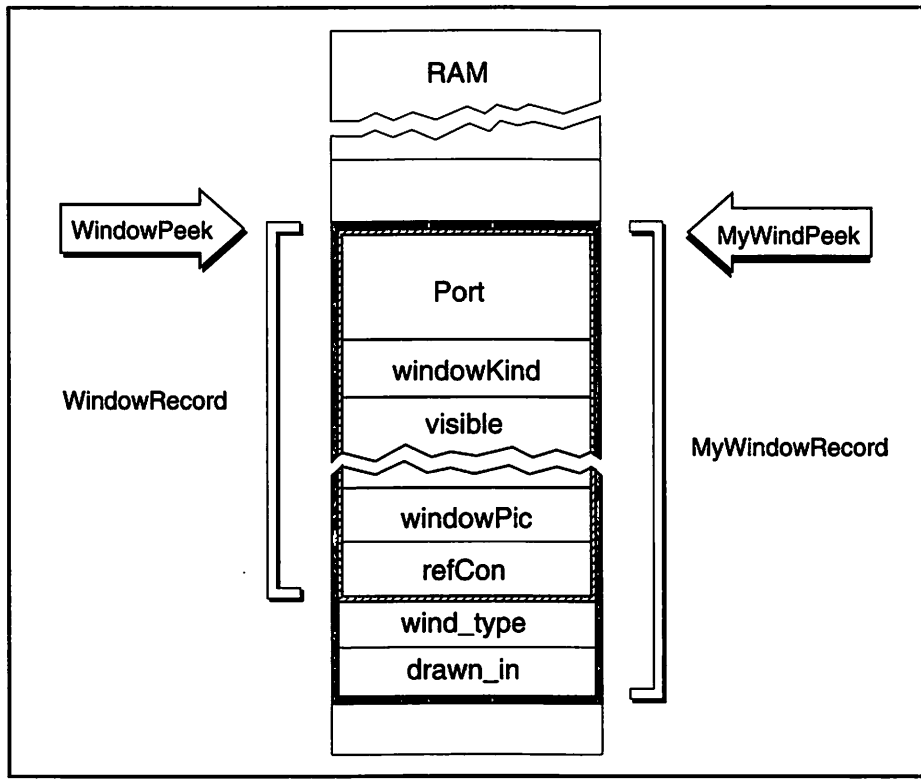


Figure 5-9. *The difference between WindowPeek and MyWindPeek*

Up to this point in your programming endeavors you've allowed the Window Manager to assign the memory storage for a window when you've called *GetNewWindow()*. You did so by passing a nil pointer as the second parameter:

```
#define     NIL                     0L
#define     IN_FRONT    (WindowPtr)-1
#define     WIND_ID                 128

WindowPtr   the_window;

the_window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
```

In a call to *GetNewWindow()* you have the option of telling the Window Manager what memory to use to store the new window. That will be a

golden opportunity to use your own structure rather than the Macintosh *WindowRecord*. Here's how it's done:

```
WindowPtr   new_window;
Ptr         wind_storage;
short       left, top;

wind_storage = NewPtr( sizeof ( MyWindRecord ) );
new_window = GetNewWindow( DRAW_WIND_ID, wind_storage, IN_FRONT );
```

Before calling *GetNewWindow()* you declare a pointer variable called *wind_storage*. This is a normal pointer, not a *WindowPtr*. Consequently it will meet the requirement that the second parameter to *GetNewWindow()* be a *Ptr*. Use *NewPtr()* to allocate a block of memory and return a pointer to it. The size of the block? Why, the size of the *MyWindRecord* data structure, of course.

Finally, make a call to *GetNewWindow()*. This time, instead of passing a nil pointer and letting the Window Manager set up the window storage, pass the *wind_storage* pointer that points to your own block of memory.

The only thing left to know is how to go about accessing the additional information that a window contains. First declare a variable to be of type MyWindPeek. Then set it to point, or peek, at a window by typecasting the window's pointer. With that accomplished you can examine and assign values to the structure members. Below is a code fragment that should help you.

```
WindowPtr    new_window;
WindowPtr    the_window;
Ptr          wind_storage;
short        left, top;
MyWindPeek   wind_peek;

wind_storage = NewPtr( sizeof ( MyWindRecord ) );
new_window   = GetNewWindow( DRAW_WIND_ID, wind_storage, IN_FRONT );
wind_peek    = ( MyWindPeek )new_window;

/* wind_peek was just set to peek at this newly created window.    */
/* Assigning a value to a member of the structure wind_peek points */
/* to only effects this one window.                                */
```

```
wind_peek->drawn_in = TRUE;

/* Later in the program you can check to see if a window, any      */
/* drawing window, has a drawing in it by checking it's personal   */
/* drawn_in member. First assign wind_peek to point to the_window, */
/* then check the member.                                          */

wind_peek  = ( MyWindPeek )the_window;

if ( wind_peek->drawn_in == TRUE )      /* examine a member        */
  [ do something here ]
```

This code "reuses" *wind_peek*. It first assigns it by having it point to a newly created window. Later it assigns it to *the_window*. You can assume in between this code some action took place to assign *the_window* to point to one of the drawing windows.

The method just described will be the backbone of the sample program at the end of this chapter. It also can be a technique you use in any of your own multiple-window programs.

This technique is one way to manage multiple window types in an application. Another approach is to store a value in the refcon field of the *WindowRecord*. The *refcon* field is a holder for any user-defined 32-bit value. Consult *Inside Macintosh* if this approach interests you.

Activates and multiple windows

Clicking the mouse on a window obscured by another window triggers the occurrence of an activate event. The clicked-on window appears to be brought to the forefront by a change in the highlighting of the window's frame. For single window programs, activate events usually aren't significant. For multiple window programs, they may be.

You can use an activate event to keep track of the most recently clicked-on, or active, window. In the example you could use a global *WindowPtr* variable for this purpose. When the user next clicks on the

control window and clicks the drawing button, action will take place in whichever window global variable *Current_Draw_Window* is pointing to. With that in mind, look at one way to handle an activate event.

```
#define    CONTROL_WINDOW    1     /* Two types of windows in this */
#define    DRAW_WINDOW       2     /* example program.             */

EventRecord  The_Event;
WindowPtr    Current_Draw_Window;  /* Save current window globally */


void  Handle_Activate( void )
{
   WindowPtr   the_window;          /* Window that was activated    */
   MyWindPeek  wind_peek;           /* Access to the window type    */
   short       window_type;

   the_window  = ( WindowPtr )The_Event.message;

   wind_peek   = ( MyWindPeek )the_window;  /* Cast to a MyWindPeek */
   window_type = wind_peek->wind_type;

   if ( window_type == DRAW_WINDOW )
      Current_Draw_Window = the_window;
}
```

As it does for updates, the Event Manager places a pointer to the window that is being activated in the *message* element of the *EventRecord*. After you have a pointer to this window, typecast it to a variable of *MyWindPeek* type so that you can access the *wind_type* element. If the window that is activated is a drawing window, set the global *Current_Draw_Window* to point to it. That way you always know which drawing window was the last one activated. Figure 5–10 summarizes the program flow from the start of the event to the pointer that's set to point to the activated window.

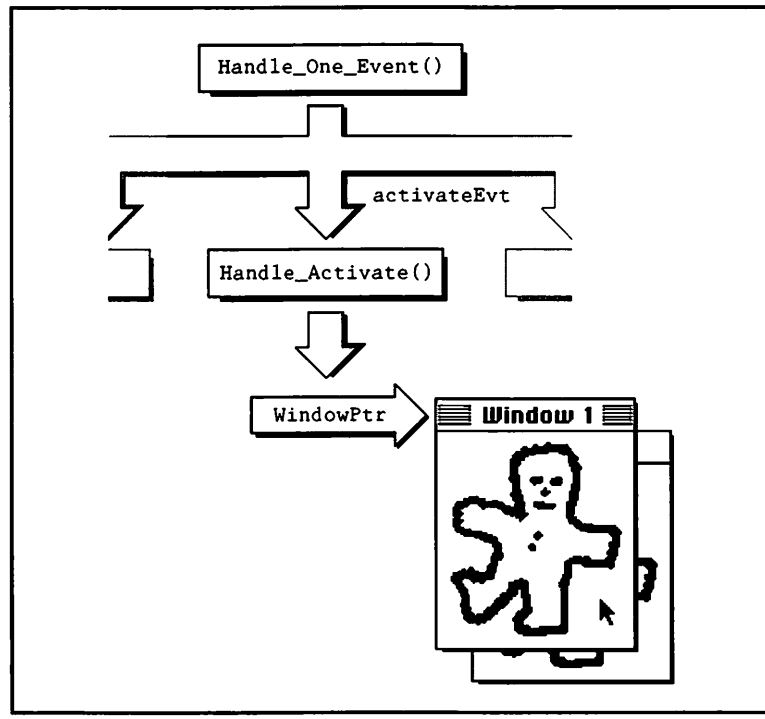


Figure 5–10. *Using an activate event to keep track of the current window*

Updates and multiple windows

All of the window updating information you read for single window programs applies to programs with more than one window. If you have different types of windows, as our example does, you'll want to have separate routines to update each. *Handle_Update()* then becomes a branching point:

```
#define    CONTROL_WINDOW    1     /* Two types of windows in this */
#define    DRAW_WINDOW       2     /* example program.             */

EventRecord  The_Event;

void  Handle_Update( void )
{
   WindowPtr   the_window;          /* Window to update            */
   MyWindPeek  wind_peek;           /* Access to the window type   */
```

```
    short       window_type;

    the_window  = ( WindowPtr )The_Event.message;

    wind_peek   = ( MyWindPeek )the_window;  /* Cast to a MyWindPeek */
    window_type = wind_peek->wind_type;

    if ( window_type == DRAW_WINDOW )
       Update_Draw_Window( the_window );
    else
       Update_Control_Window( the_window );
}
```

Both update routines begin by getting and saving the ports. They then, nested between calls to *BeginUpdate()* and *EndUpdate()*, perform all the text and graphics drawing tasks necessary for a window. Figure 5–11 shows updating when more than one window is present.

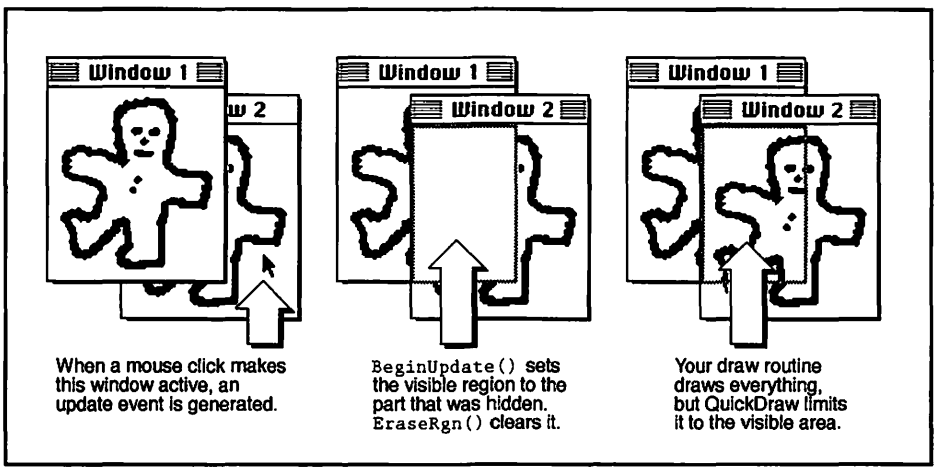
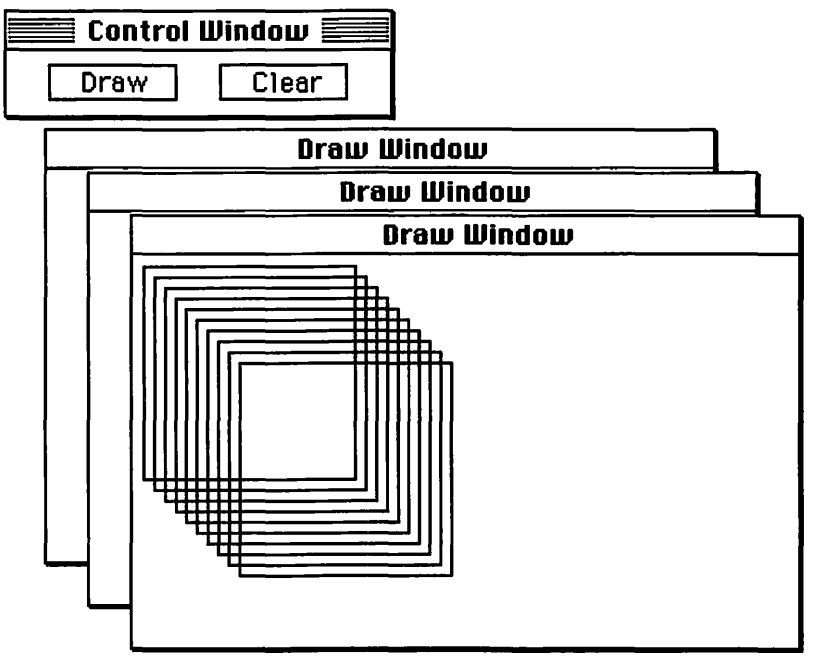


Figure 5–11. *Updating a window in a multiple-window program*

Chapter Program: Working With Multiple Windows

As a working example of the multiple window techniques just discussed, this chapter presents *MultiWindows*—a program that is capable of displaying multiple windows. *MultiWindows* will put two types of

windows on the screen: a control window and a drawing window. Additionally, the program will open more than one drawing window. Figure 5–12 is a screen shot of the windows you'll see when you run *MultiWindows*.



5–12. MultiWindows *program in action*

MultiWindows allows the user to draw a pattern or erase an existing pattern in any one of the three drawing windows. The last drawing window selected will be the window where the action takes place.

Program resources: *MultiWindows.π.rsrc*

The *MultiWindows* program has just two resources, both 'WIND's. The 'WIND' with ID 128 will be the control window, while 'WIND' 129 will serve as the template for each of the drawing windows. Figure 5–13 shows the two resources.

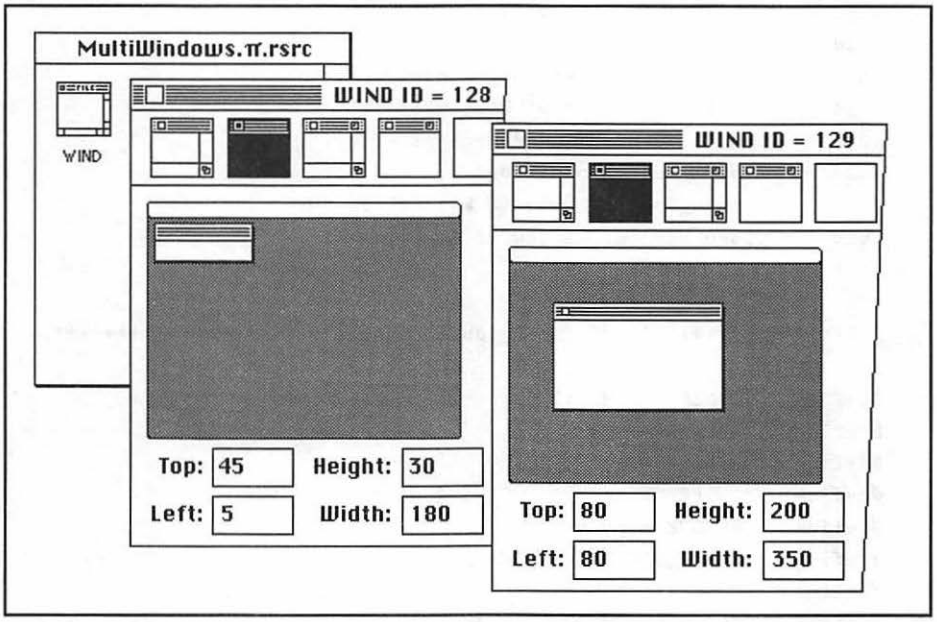


Figure 5-13. *The two 'WIND' resources for* MultiWindows

Program listing: *MultiWindows.c*

Here is the code in its entirety. A description follows.

```
/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include <Traps.h>


/*++++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void     Initialize_Toolbox( void );
void     Set_Window_Drag_Boundaries( void );
void     Initialize_Variables( void );
void     Set_Window_Type( WindowPtr, short );
short    Determine_Window_Type( WindowPtr );
void     Set_Drawn_In_Flag( WindowPtr, Boolean );
Boolean  Determine_Drawn_In_Flag( WindowPtr );
void     Open_Control_Window( void );
void     Open_Draw_Window( void );
void     Handle_One_Event( void );
```

```
void      Handle_Activate( void );
void      Handle_Update( void );
void      Update_Control_Window( WindowPtr );
void      Update_Draw_Window( WindowPtr );
void      Draw_Something( WindowPtr );
void      Handle_Mouse_Down( void );
void      Handle_Control_Window( WindowPtr, Point );
void      Close_Window( WindowPtr );


/*++++++++++++++++++   Define global constants   ++++++++++++++++++*/

#define    CONTROL_WIND_ID               128
#define    DRAW_WIND_ID                  129
#define    NIL                            0L
#define    IN_FRONT           (WindowPtr)-1L
#define    REMOVE_EVENTS                  0
#define    SLEEP_TICKS                   0L
#define    MOUSE_REGION                  0L
#define    CONTROL_WINDOW                 1
#define    DRAW_WINDOW                    2
#define    STR_LIST_ID                  128
#define    CONTROL_WIND_TITLE_STR         1
#define    DRAW_WIND_TITLE_STR            2
#define    DRAW_BUTTON_STR                3
#define    CLEAR_BUTTON_STR               4
#define    WIND_LEFT                     30
#define    WIND_TOP                     100
#define    WIND_OFFSET                   20
#define    DRAG_EDGE                     10


/*++++++++++++++++++   Define global types   ++++++++++++++++++*/

typedef  struct
{
   WindowRecord   wind_rec;
   short          wind_type;
   Boolean        drawn_in;
}  MyWindRecord, *MyWindPeek;


/*++++++++++++++++   Define global variables   ++++++++++++++++*/

Boolean      All_Done = FALSE;
```

```
Boolean      Multifinder_Present;
EventRecord  The_Event;
Rect         Drag_Rect;
WindowPtr    Current_Draw_Window;
short        Window_Type;
Rect         Draw_Rect;
Rect         Clear_Rect;
short        Num_Draw_Winds_Open;


/*++++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   Initialize_Toolbox();
   Set_Window_Drag_Boundaries();
   Initialize_Variables();

   Open_Control_Window();
   Open_Draw_Window();
   Open_Draw_Window();
   Open_Draw_Window();

   while ( All_Done == FALSE )
      Handle_One_Event();
}


/*++++++++++++++++++  Initialize the Toolbox  ++++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*++++++++++++  Initialize window drag boundaries  ++++++++++++*/
```

```
void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = screenBits.bounds;
   Drag_Rect.left += DRAG_EDGE;
   Drag_Rect.right -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}


/*+++++++++++++++  Initialize some of our variables  ++++++++++++*/

void  Initialize_Variables( void )
{
   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));

   Num_Draw_Winds_Open = 0;

   SetRect( &Draw_Rect, 20, 6, 80, 23 );
   SetRect( &Clear_Rect, 100, 6, 160, 23 );
}


/*++++++++++++  Set a window's type: control or draw  ++++++++++*/

void  Set_Window_Type(WindowPtr the_window, short type)
{
   MyWindPeek   wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   wind_peek->wind_type = type;
}


/*+++++++++  Examine a window's type: control or draw  +++++++++*/

short  Determine_Window_Type( WindowPtr the_window )
{
   MyWindPeek   wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   return ( wind_peek->wind_type );
}
```

```
/*++++  Set a window's drawn flag: does it have a drawing?  ++++*/

void  Set_Drawn_In_Flag( WindowPtr the_window, Boolean drawn )
{
   MyWindPeek   wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   wind_peek->drawn_in = drawn;
}


/*++  Examine a window's drawn flag: does it have a drawing?  ++*/

Boolean  Determine_Drawn_In_Flag( WindowPtr the_window )
{
   MyWindPeek  wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   return ( wind_peek->drawn_in );
}


/*++++++++++++++++++  Open one control window  ++++++++++++++++++*/

void  Open_Control_Window( void )
{
   WindowPtr  new_window;
   Ptr        wind_storage;
   Str255     the_str;

   wind_storage = NewPtr( sizeof( MyWindRecord ) );
   new_window  = GetNewWindow( CONTROL_WIND_ID, wind_storage, IN_FRONT
);

   if ( new_window == NIL )
      ExitToShell();

   Set_Drawn_In_Flag( new_window, FALSE );
   Set_Window_Type( new_window, CONTROL_WINDOW );

   GetIndString( the_str, STR_LIST_ID, CONTROL_WIND_TITLE_STR );
   SetWTitle( new_window, the_str );
   ShowWindow( new_window );
}
```

```
/*++++++++++++++++++  Open one drawing window  ++++++++++++++++++*/

void  Open_Draw_Window( void )
{
   WindowPtr  new_window;
   Ptr        wind_storage;
   short      left, top;
   Str255     the_str;

   wind_storage = NewPtr( sizeof ( MyWindRecord ) );
   new_window   = GetNewWindow( DRAW_WIND_ID, wind_storage, IN_FRONT );

   if ( new_window == NIL )
      ExitToShell();

   Set_Drawn_In_Flag( new_window, FALSE );

   Set_Window_Type( new_window, DRAW_WINDOW );

   GetIndString( the_str, STR_LIST_ID, DRAW_WIND_TITLE_STR );
   SetWTitle( new_window, the_str );
   left =  WIND_LEFT + ( Num_Draw_Winds_Open * WIND_OFFSET );
   top  =  WIND_TOP  + ( Num_Draw_Winds_Open * WIND_OFFSET );
   MoveWindow( new_window, left, top, TRUE );
   ShowWindow( new_window );

   Num_Draw_Winds_Open++;
}


/*++++++++++++++++++++  Handle a single event  ++++++++++++++++++*/

void  Handle_One_Event( void )
{
   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
```

```
      case mouseDown:
         Handle_Mouse_Down();
         break;

      case updateEvt:
         Handle_Update();
         break;

      case activateEvt:
         Handle_Activate();
         break;
   }
}


/*+++++++++++++++++  Handle an activate event  +++++++++++++++++*/

void  Handle_Activate( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;
   Window_Type = Determine_Window_Type( the_window );

   if ( Window_Type == DRAW_WINDOW )
      Current_Draw_Window = the_window;
}


/*+++++++++++++++++++  Handle an update event  +++++++++++++++++*/

void  Handle_Update( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;
   Window_Type = Determine_Window_Type( the_window );

   if ( Window_Type == DRAW_WINDOW )
      Update_Draw_Window( the_window );
   else
      Update_Control_Window( the_window );
}
```

```
/*++++++++++++++++++  Update the control window  +++++++++++++++++*/

void  Update_Control_Window( WindowPtr the_window )
{
   GrafPtr  old_port;
   Str255   the_str;

   GetPort( &old_port );
   SetPort( the_window );
   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      FrameRect( &Draw_Rect );
      MoveTo( Draw_Rect.left + 15, Draw_Rect.bottom - 4 );
      GetIndString( the_str, STR_LIST_ID, DRAW_BUTTON_STR );
      DrawString( the_str );
      FrameRect( &Clear_Rect);
      MoveTo( Clear_Rect.left + 15, Clear_Rect.bottom - 4 );
      GetIndString( the_str, STR_LIST_ID, CLEAR_BUTTON_STR );
      DrawString( the_str );
   EndUpdate( the_window );
   SetPort( old_port );
}


/*++++++++++++++++++  Update a drawing window  +++++++++++++++++*/

void  Update_Draw_Window( WindowPtr the_window )
{
   GrafPtr  old_port;

   GetPort( &old_port );
   SetPort( the_window );
   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      if ( Determine_Drawn_In_Flag( the_window ) )
         Draw_Something( the_window );
   EndUpdate( the_window );

   SetPort( old_port );
}


/*++++++++++++++  Draw something to a draw window  +++++++++++++*/

void  Draw_Something( WindowPtr the_window )
```

```
{
   GrafPtr  old_port;
   Rect     the_rect;
   short    i;

   GetPort( &old_port );
   SetPort( the_window );

   for ( i=1; i <= 10; i++ )
   {
      SetRect( &the_rect, i*5, i*5, i*5+100, i*5+100 );
      FrameRect( &the_rect );
   }

   SetPort( old_port );
}


/*++++++++++++  Handle a click of the mouse button  ++++++++++++++*/

void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         break;

      case inSysWindow:
         SystemClick( &The_Event, the_window );
         break;

      case inDrag:
         DragWindow( the_window, The_Event.where, &screenBits.bounds );
         break;

      case inGoAway:
         if ( TrackGoAway( the_window, The_Event.where ) )
            Close_Window( the_window );
         break;
```

```
      case inContent:
         if ( the_window != FrontWindow() )
            SelectWindow( the_window );
         else
         {
            Window_Type = Determine_Window_Type( the_window );
            if ( Window_Type == CONTROL_WINDOW )
               Handle_Control_Window( the_window, The_Event.where );
         }
         break;
   }
}


/*++++++++++  Handle a click in the control window  ++++++++++++*/

void  Handle_Control_Window( WindowPtr the_window, Point the_point )
{
   GrafPtr  old_port;

   SetPort( the_window );
   GlobalToLocal( &the_point );

   if ( PtInRect( the_point, &Draw_Rect ) )
   {
      InvertRect( &Draw_Rect );
      Draw_Something( Current_Draw_Window );
      Set_Drawn_In_Flag( Current_Draw_Window, TRUE );
      InvertRect( &Draw_Rect );
   }
   if ( PtInRect( the_point, &Clear_Rect ) )
   {
      InvertRect( &Clear_Rect );
      GetPort( &old_port );
      SetPort( Current_Draw_Window );
      EraseRect( &Current_Draw_Window->portRect );
      Set_Drawn_In_Flag( Current_Draw_Window, FALSE );
      SetPort( old_port );
      InvertRect( &Clear_Rect );
   }
}
```

```
/*++++++++++++++++++   Close one window   ++++++++++++++++++++*/

void  Close_Window( WindowPtr the_window )
{
   HideWindow( the_window );
   CloseWindow( the_window );
   DisposPtr( ( Ptr )the_window );
   All_Done = TRUE;
   Num_Draw_Winds_Open--;
}
```

Stepping through the code

Now let's walk through the MultiWindows code, placing emphasis on the new material.

The #include directives

MultiWindows uses the *Traps.h #include* file. *Traps.h* contains information that will be used in the call to *NGetTrapAddress()*. Chapter 8 has more to say about the *Traps.h* file.

The #define directives

MultiWindows opens two types of windows, each defined by a 'WIND' resource template. Their resource IDs are *CONTROL_WIND_ID* and *DRAW_WIND_ID*. Both *NIL* and *IN_FRONT* are parameters for *GetNewWindow()*. *REMOVE_EVENTS* is used during initialization. A call to *WaitNextEvent()* uses *SLEEP_TICKS* and *MOUSE_REGION* as parameters. To distinguish between the two window types, *MultiWindows* calls one a *CONTROL_WINDOW* and the other a *DRAW_WINDOW*. The titles to the two window types and the two buttons are in a string list resource. *CONTROL_WIND_TITLE_STR*, *DRAW_WIND_TITLE_STR*, *DRAW_BUTTON_STR*, and *CLEAR_BUTTON_STR* serve as indices to the 'STR#' resource with an ID of *STR_LIST_ID*. After loading a drawing window the program moves it on the screen to stagger it from other open windows. *WIND_LEFT*, *WIND_TOP*, and *WIND_OFFSET* help there. *DRAG_RECT* gives a pixel buffer that prevents a window from going off screen.

```
#define    CONTROL_WIND_ID              128
#define    DRAW_WIND_ID                 129
#define    NIL                           0L
#define    IN_FRONT          (WindowPtr)-1L
#define    REMOVE_EVENTS                 0
#define    SLEEP_TICKS                   0L
#define    MOUSE_REGION                  0L
#define    CONTROL_WINDOW                1
#define    DRAW_WINDOW                   2
#define    STR_LIST_ID                 128
#define    CONTROL_WIND_TITLE_STR        1
#define    DRAW_WIND_TITLE_STR           2
#define    DRAW_BUTTON_STR               3
#define    CLEAR_BUTTON_STR              4
#define    WIND_LEFT                    30
#define    WIND_TOP                    100
#define    WIND_OFFSET                  20
#define    DRAG_EDGE                    10
```

Global types

This chapter spent a lot of time going over a strategy that would allow a program to be able to distinguish one type of window from another. The *MyWindRecord* struct and the *MyWindPeek* that point to it are defined here exactly as they were earlier in this chapter.

```
typedef  struct
{
   WindowRecord   wind_rec;
   short          wind_type;
   Boolean        drawn_in;
}  MyWindRecord, *MyWindPeek;
```

Global variables

MultiWindows handles events as described at the start of this chapter, making use of *All_Done*, *Multifinder_Present*, and *The_Event*. The program sets up *Drag_Rect* to prevent a window from disappearing off the screen. The program uses *Current_Draw_Window* to keep track of the window that was clicked on last. I've used variable *Num_Draw_Windows_Open* to stagger the drawing windows as they open. The control window has two rectangles

that serve as buttons. The *Draw_Rect* and *Clear_Rect* variables hold the boundaries of these rectangles.

```
Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
Rect         Drag_Rect;
WindowPtr    Current_Draw_Window;
short        Window_Type;
Rect         Draw_Rect;
Rect         Clear_Rect;
short        Num_Draw_Winds_Open;
```

The main() function

Like most good *main()* functions, this one is short and simple. It first calls three initialization routines. You saw *Initialize_Toolbox()* in the last chapter and *Set_Window_Drag_Rectangle()* in this chapter. The *Initialize_Variables()* routine simply groups together some miscellaneous one-time assignments.

MultiWindows opens four windows—one control window and three drawing windows. The best way to display the windows is to put the control window on the screen and then let the user select as many drawing windows as he wants by making a menu selection. I chose this method because I want to keep things simple. The purpose of this program is to demonstrate window handling, so I've kept the code to a minimum by omitting features (like menus) you'd be sure to have in a full Macintosh application. Look in Chapter 9 for an example that demonstrates a complete program making use of all the features and techniques discussed in this book.

The *main()* routine ends with the ever-faithful *while* loop that drives the program.

```
void  main( void )
{
   Initialize_Toolbox();
   Set_Window_Drag_Rectangle();
   Initialize_Variables();

   Open_Control_Window();
   Open_Draw_Window();
   Open_Draw_Window();
   Open_Draw_Window();

   while ( All_Done == FALSE )
      Handle_One_Event();
}
```

Initialization

Most of the program initialization stuff should look familiar to you by now. The two global *Rect* variables, *Draw_Rect* and *Clear_Rect*, are given their boundaries in *Initialize_Variables*. Later in the program you'll frame these two rectangles in the control window and use them as buttons.

```
void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


void  Set_Window_Drag_Rectangle( void )
{
   Drag_Rect = screenBits.bounds;
   Drag_Rect.left += DRAG_EDGE;
   Drag_Rect.right -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}
```

```
void  Initialize_Variables( void )
{
   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));

   Num_Draw_Winds_Open = 0;

   SetRect( &Draw_Rect, 20, 6, 80, 23 );
   SetRect( &Clear_Rect, 100, 6, 160, 23 );
}
```

Marking and examining a window

This chapter's Multiple Window Techniques section worked out a strategy for adding information to a window so that it can contain more data than a *WindowRecord* alone. *MultiWindows* makes full use of this technique. When the program creates a window, *Set_Window_Type()* is called. This routine receives a pointer to the new window and then marks the window as one of the program's two types, depending on the passed-in value type. To access the *wind_type*, cast the *WindowPtr* variable to a *MyWindPeek* variable. Figure 5–14 shows what happens with this typecasting.

```
void  Set_Window_Type(WindowPtr the_window, short type)
{
   MyWindPeek   wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   wind_peek->wind_type = type;
}
```

Note that the typecasting of a *WindowPtr* variable to a *MyWindPeek* variable must occur only with a *WindowPtr* variable that you are sure points to a *MyWindRecord* structure. Otherwise, *wind_peek->wind_type* will access unrelated memory that lies beyond the end of the *WindowRecord* structure!

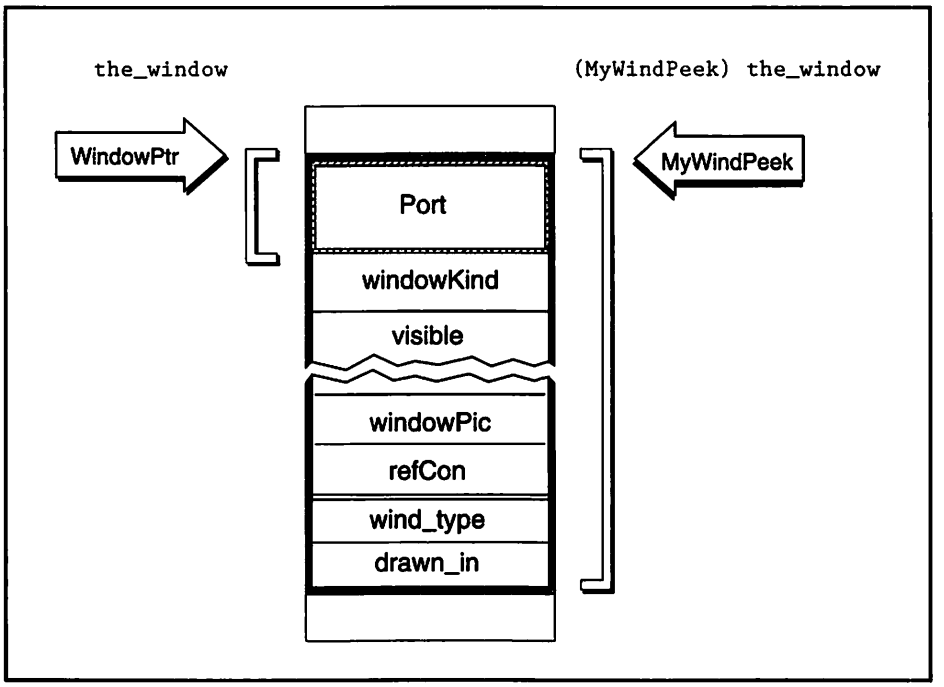


Figure 5-14. *Typecasting a* WindowPtr *to a* MyWindPeek

If you understand *Set_Window_Type()*, you'll understand the next three routines. Instead of setting a window's type, *Determine_Window_Type()* examines the *wind_type*. It then returns the type so that the program can make a decision based on the this information.

```
short  Determine_Window_Type( WindowPtr the_window )
{
   MyWindPeek   wind_peek;

   wind_peek  = ( MyWindPeek )the_window;
   return ( wind_peek->wind_type );
}
```

The *Set_Drawn_In_Flag()* and *Determine_Drawn_In_Flag()* work in the same way as the preceding two routines.

```
void  Set_Drawn_In_Flag( WindowPtr the_window, Boolean drawn )
{
```

```
    MyWindPeek   wind_peek;

    wind_peek  = ( MyWindPeek )the_window;
    wind_peek->drawn_in = drawn;
}


Boolean  Determine_Drawn_In_Flag( WindowPtr the_window )
{
    MyWindPeek  wind_peek;

    wind_peek  = ( MyWindPeek )the_window;
    return ( wind_peek->drawn_in );
}
```

Opening a window

When *MultiWindows* opens a new window it reserves memory the size of *MyWindRecord* rather than the size of the Macintosh C type *WindowRecord*. This allows it to store the type of the window and a flag that tells whether the window has a drawing in it. *Open_Control_Window()* does all of that. It also changes the title that appears in the window's title bar from "Untitled" to the more descriptive title stored in the program's 'STR#' resource.

```
void  Open_Control_Window( void )
{
    WindowPtr  new_window;
    Ptr        wind_storage;
    Str255     the_str;

    wind_storage = NewPtr( sizeof( MyWindRecord ) );
    new_window   = GetNewWindow( CONTROL_WIND_ID, wind_storage, IN_FRONT );

    if ( new_window == NIL )
        ExitToShell();

    Set_Drawn_In_Flag( new_window, FALSE );
    Set_Window_Type( new_window, CONTROL_WINDOW );

    GetIndString( the_str, STR_LIST_ID, CONTROL_WIND_TITLE_STR );
    SetWTitle( new_window, the_str );
    ShowWindow( new_window );
}
```

Opening a drawing window involves all of the same steps as opening a control window, and a few more. Because more than one drawing window will be open, you'll want to keep track of which one is active. What about setting that global window pointer *Current_Draw_Window* to point to this new window? If it just opened, it surely must be the active, current window, right? Right. But you don't have to take care of that here. A mouse click on an obscured draw window will also trigger an activate event, so both a click on a window and the opening of a new window will lead the program to *Handle_Activate()*. If you update *Current_Draw_Window* in your *Handle_Activate()* routine, you're assured of keeping that variable pointing at the right window no matter how a window gets activated.

Open_Draw_Window() finishes up by setting the window's title, then prettying things up a little by offsetting the window from any other newly opened drawing windows. It uses the number of open windows in the calculation of the location for the new window. The more windows that are open, the greater the offset will be.

```
void  Open_Draw_Window( void )
{
   WindowPtr  new_window;
   Ptr        wind_storage;
   short      left, top;
   Str255     the_str;

   wind_storage = NewPtr( sizeof ( MyWindRecord ) );
   new_window   = GetNewWindow( DRAW_WIND_ID, wind_storage, IN_FRONT );

   if ( new_window == NIL )
      ExitToShell();

   Set_Drawn_In_Flag( new_window, FALSE );

   Set_Window_Type( new_window, DRAW_WINDOW );

   GetIndString( the_str, STR_LIST_ID, DRAW_WIND_TITLE_STR );
   SetWTitle( new_window, the_str );
   left =  WIND_LEFT + ( Num_Draw_Winds_Open * WIND_OFFSET );
   top  =  WIND_TOP  + ( Num_Draw_Winds_Open * WIND_OFFSET );
   MoveWindow( new_window, left, top, TRUE );
   ShowWindow( new_window );
```

```
    Num_Draw_Winds_Open++;
}
```

Event handling

MultiWindows starts the handling of an event by calling *Handle_One_Event()*. The program responds to three types of events: *mouseDown*, *updateEvt*, and *activateEvt*.

```
void  Handle_One_Event( void )
{
   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
      case mouseDown:
         Handle_Mouse_Down();
         break;

      case updateEvt:
         Handle_Update();
         break;

      case activateEvt:
         Handle_Activate();
         break;
   }
}
```

There is only one task that *Handle_Activate()* is responsible for handling: setting *Current_Draw_Window* to point to the activated window. Provided, of course, the window is a drawing window.

```
void  Handle_Activate( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;
   Window_Type = Determine_Window_Type( the_window );

   if ( Window_Type == DRAW_WINDOW )
      Current_Draw_Window = the_window;
}
```

If the event is an update event, *MultiWindows* determines the type of window the update is for. It then branches to the correct routine for further processing.

```
void  Handle_Update( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;
   Window_Type = Determine_Window_Type( the_window );

   if ( Window_Type == DRAW_WINDOW )
      Update_Draw_Window( the_window );
   else
      Update_Control_Window( the_window );
}
```

The control window is updated by redrawing the two rectangles that serve as its buttons. Since drawing is taking place, call *SetPort()* to make the control window's port current.

```
void  Update_Control_Window( WindowPtr the_window )
{
   GrafPtr  old_port;
   Str255   the_str;

   GetPort( &old_port );
   SetPort( the_window );
   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      FrameRect( &Draw_Rect );
```

```
        MoveTo( Draw_Rect.left + 15, Draw_Rect.bottom - 4 );
        GetIndString( the_str, STR_LIST_ID, DRAW_BUTTON_STR );
        DrawString( the_str );
        FrameRect( &Clear_Rect);
        MoveTo( Clear_Rect.left + 15, Clear_Rect.bottom - 4 );
        GetIndString( the_str, STR_LIST_ID, CLEAR_BUTTON_STR );
        DrawString( the_str );
    EndUpdate( the_window );
    SetPort( old_port );
}
```

Speaking of drawing buttons, did you notice that *MultiWindows* didn't explicitly draw them when the control window was opened back in *Open_Control_Window()*? Yet, the buttons were drawn at that time. That's because *GetNewWindow()* highlights the new window, then causes both an activate and update event to occur. *Handle_Activate()* doesn't do anything related to the control window, but *Handle_Update()* calls *Update_Control_Window()*, which then draws and labels the rectangles. Figure 5–15 illustrates this.

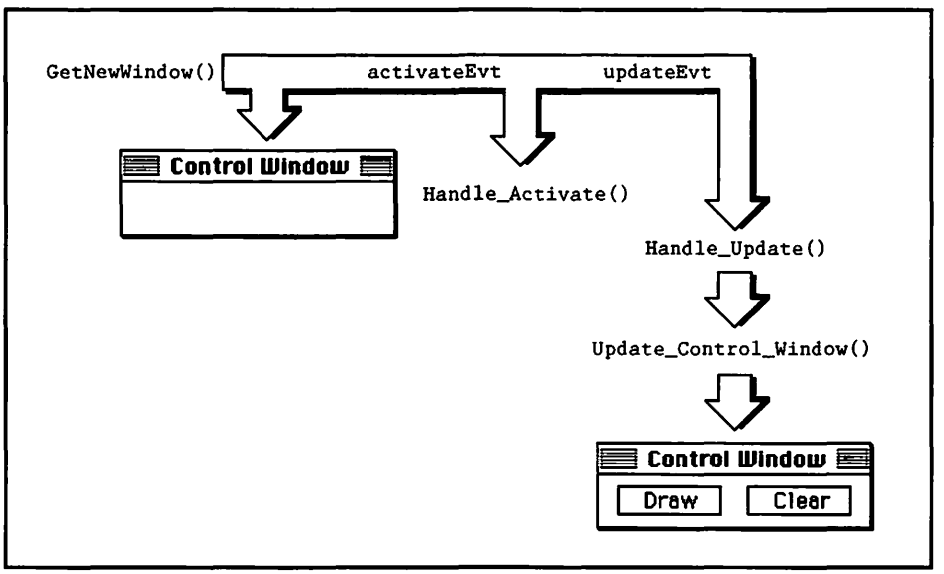


Figure 5–15. *GetNewWindow triggers two events*

When a new drawing window is opened, or an existing window is activated or moved from off screen to on screen, an update event occurs, and *Update_Draw_Window()* is called. This routine checks the *drawn_in*

flag of the window that needs updating to see if a drawing is present. If a drawing is present, call *Draw_Something()* to redraw the graphics. Because drawing might take place, set the port and call *BeginUpdate()* and *EndUpdate()*.

```
void  Update_Draw_Window( WindowPtr the_window )
{
   GrafPtr  old_port;

   GetPort( &old_port );
   SetPort( the_window );
   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      if ( Determine_Drawn_In_Flag( the_window ) )
         Draw_Something( the_window );
   EndUpdate( the_window );

   SetPort( old_port );
}
```

Earlier in this chapter I used a routine called *Draw_Something()* to load a 'PICT' resource and display it to a window. Here, I let QuickDraw frame ten overlapping rectangles from within a loop. You'll be able to come up with something much more interesting for your own program. For these examples, simplicity rules.

```
void  Draw_Something( WindowPtr the_window )
{
   GrafPtr  old_port;
   Rect     the_rect;
   short    i;

   GetPort( &old_port );
   SetPort( the_window );

   for ( i=1; i <= 10; i++ )
   {
      SetRect( &the_rect, i*5, i*5, i*5+100, i*5+100 );
      FrameRect( &the_rect );
   }

   SetPort( old_port );
}
```

A click of the mouse is the third and final type of event *MultiWindows* handles. Since there are no menus, the program ignores a click in the menu. The appropriate Toolbox calls handle a mouse click in a desk accessory or a window's drag bar. A click in the close box of a drawing window invokes *Close_Window()*, which I'll discuss in a moment.

A mouse click in the content of a window warrants more discussion. If the window was not active before the click, call *SelectWindow()* and consider the event handled. If the window is already active, make a check to see if the window is the control window. If so, you'll want to determine if the click of the mouse button is in one of the two rectangles. *The_Event.where* holds the screen pixel coordinates of the mouse-button click, so pass that value to *Handle_Control_Window()*. a routine to further process it.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         break;

      case inSysWindow:
         SystemClick( &The_Event, the_window );
         break;

      case inDrag:
         DragWindow( the_window, The_Event.where, &screenBits.bounds );
         break;

      case inGoAway:
         if ( TrackGoAway( the_window, The_Event.where ) )
            Close_Window( the_window );
         break;

      case inContent:
         if ( the_window != FrontWindow() )
```

```
            SelectWindow( the_window );
         else
         {
            Window_Type = Determine_Window_Type( the_window );
            if ( Window_Type == CONTROL_WINDOW )
               Handle_Control_Window( the_window, The_Event.where );
         }
         break;
   }
}
```

Handle_Control_Window() uses a Toolbox routine called *GlobalToLocal()* to determine if a mouse click occurred in either of the control window's two rectangles. The Point value passed into *Handle_Control_Window()*, *The_Event.where*, is in global, or screen coordinates. Drawing in a window takes place in local, or window coordinates. Figure 5–16 should clarify this point.

The Toolbox routine *PtInRect()* returns a value of true if the passed-in *Point* variable lies in the passed-in rectangle. If the mouse click is in the drawing rectangle, invert the rectangle to let the user know his click was registered. Then examine the window's *drawn_in* flag. If the last drawing window to be active already has a drawing in it, you won't want to bother drawing it again. If not, call *Draw_Something()* to make the drawing. Then set the window's *drawn_in* flag to mark the window as having a drawing in it—vital information you'll use during window updating. When complete, invert the rectangle back to its original state.

A mouse button click in the clear button is handled in a manner similar to a click in the draw button.

```
void  Handle_Control_Window( WindowPtr the_window, Point the_point )
{
   GrafPtr  old_port;

   SetPort( the_window );
   GlobalToLocal( &the_point );

   if ( PtInRect( the_point, &Draw_Rect ) )
   {
      InvertRect( &Draw_Rect );
      if ( Determine_Drawn_In_Flag( Current_Draw_Window ) == FALSE )
```

```
        {
            Draw_Something( Current_Draw_Window );
            Set_Drawn_In_Flag( Current_Draw_Window, TRUE );
        }
        InvertRect( &Draw_Rect );
    }
    if ( PtInRect( the_point, &Clear_Rect ) )
    {
        InvertRect( &Clear_Rect );
        if ( Determine_Drawn_In_Flag( Current_Draw_Window ) == TRUE )
        {
            GetPort( &old_port );
            SetPort( Current_Draw_Window );
            EraseRect( &Current_Draw_Window->portRect );
            Set_Drawn_In_Flag( Current_Draw_Window, FALSE );
            SetPort( old_port );
        }
        InvertRect( &Clear_Rect );
    }
}
```

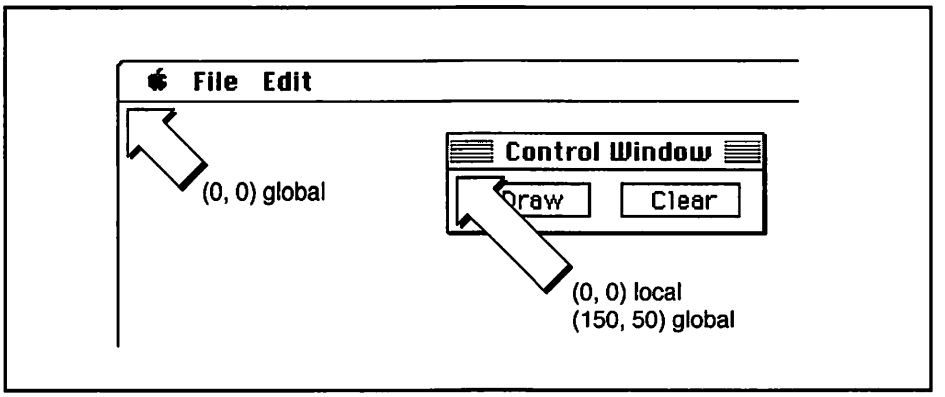


Figure 5-16. *Global and local pixel coordinates*

A click anywhere in a window closes that window. Since *MultiWindows* has no menu bar, the program uses a click in a go-away box to end the program by setting *All_Done* to true. In a real world application, you'd omit the *All_Done* line and instead set it to true when the user selected "Quit" from the program's File menu.

Notice that to close the window, you make calls to two Toolbox routines: *CloseWindow()* and *DisposPtr()*. Earlier in this chapter you

closed a window by simply calling a different Toolbox routine—*DisposeWindow()*. When you supply the window storage for *GetNewWindow()*, as you do here, call *CloseWindow()* and *DisposPtr()*. If you let the Mac handle window storage, as you do when you pass nil as the second parameter, just call *DisposeWindow()*.

```
void  Close_Window( WindowPtr the_window )
{
   HideWindow( the_window );
   CloseWindow( the_window );
   DisposPtr( ( Ptr )the_window );
   All_Done = TRUE;
   Num_Draw_Winds_Open--;
}
```

Chapter Summary

The 'WIND' resource type defines the look of a window. A call to *GetNewWindow()* loads a 'WIND' resource into memory, ready to be displayed on the screen with a call to *ShowWindow()*.

The descriptive information about a window is read in from the 'WIND' resource and, along with additional information that can be set within source code, is stored to a *WindowRecord*. Rather than access the fields of the *WindowRecord* directly, you use Toolbox routines. These Toolbox routines accept a *WindowPtr*, a pointer to the *WindowRecord*, rather than the *WindowRecord* itself.

Most window-related information that a programmer needs to access is available through use of the *WindowPtr*, which points to a window's graphics port. For those few times when you need to access other information, you'll use a *WindowPeek*. A *WindowPeek* points to the entire *WindowRecord*, rather than just to the graphics port.

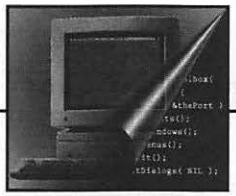
Much of the work involved in handling a window occurs when a user presses the mouse button, causing a mouse down event. When your program receives a mouse down event you'll handle it according to the location on the window where the event took place. To drag a window, you'll call *DragWindow()*. To close a window, you can first hide it with a call to

HideWindow(), then dispose of it with *DisposeWindow()*. In response to a click in the content of the window, you can call *SelectWindow()* to bring the window to the front of the screen.

When a covered window becomes exposed, you call the Toolbox routines *BeginUpdate()* and *EndUpdate()*. In between the calls, you take care of any of the drawing that needs to be done for the particular window that needs updating.

Some applications make use of windows that perform different functions. One window may accept input from the user, while another displays some graphical output. For a multiple-window application you have to use a technique that lets your application distinguish between these different windows. Failure to do so will cause window-updating problems.

One strategy for handling multiple windows is to expand the *WindowRecord*. To do this you create your own data structure that contains an entire *WindowRecord* and any additional information you want associated with a window. The primary new information will be a variable that holds the type of the window.



6 Dealing with Dialogs

The primary method of relaying information to a Macintosh program is through a dialog box. Allowing a user to adjust program settings is a typical use of a dialog box. A Macintosh program issues warnings to the user in the form of an alert, the simplest of dialog boxes.

In this chapter you'll learn how to create alerts using the 'ALRT' and 'DITL' resources. This will be the foundation for creating dialog box resources as well. Dialogs use the 'DLOG' and 'DITL' resource types.

Here you'll see the similarities between windows and dialogs. You will learn that dialog boxes are little more than embellished windows. This chapter will cover both the fixed modal dialog and the movable modeless dialog.

Finally, the example program will demonstrate all the dialog box techniques covered in this chapter, along with a method for handling the case of both a window and a dialog coexisting on the screen.

Alerts

When a program's user makes a mistake, or is about to embark on a path the program's creator feels is dangerous, the user meets with an alert. An alert provides a warning. It can strictly prohibit the impending action from taking place, or it may provide a warning and then give the user the chance to back out or carry on. Figure 6–1 shows an alert.

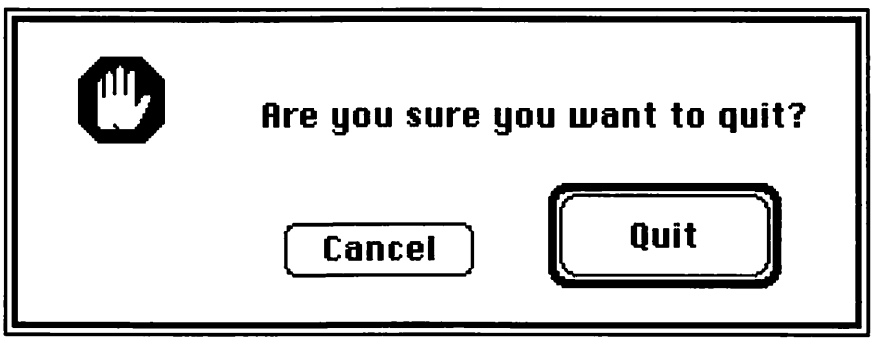


Figure 6–1. *A typical alert*

An alert typically contains text and one or two push buttons, such as the Cancel and Quit buttons in Figure 6–1. You'll need two resource types for an alert: the 'ALRT' and the 'DITL'. I cover them next.

Alert resources: 'ALRT' and 'DITL'

The 'ALRT' resource defines the size and screen placement of an alert, just as the 'WIND' resource defines the same for a window. Whereas you specify the type of window to display for a 'WIND', you don't for an 'ALRT'. An alert always has the appearance of the one pictured in 6–1.

An 'ALRT' requires that you give the ID of yet another resource—a 'DITL' that corresponds to the 'ALRT'. The 'ALRT' gives the size and placement of the alert; the 'DITL' gives the contents of the alert; the contents consist of such things as the buttons and text that are to appear in the alert. Figure 6–2 shows an 'ALRT' with an ID of 129, as can be seen in the title bar. The 'DITL' ID is also 129.

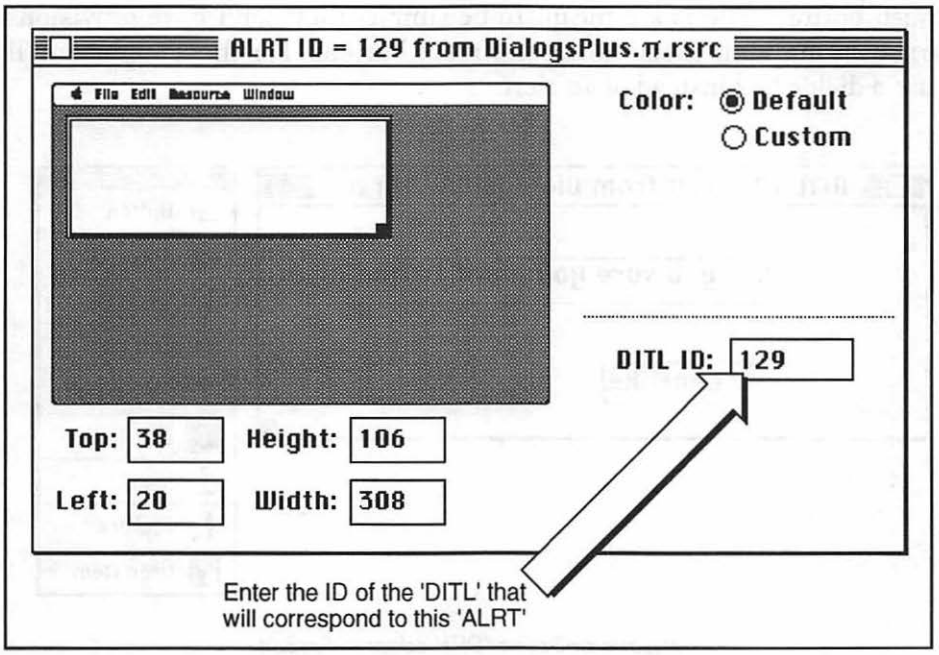


Figure 6–2. *The 'ALRT' editor in ResEdit*

The 'DITL' that corresponds to an 'ALRT' doesn't have to have the same ID as the 'ALRT', but because it makes sense to do so, programmers usually give it the same ID.

You create the 'ALRT' using the "Create New Resource" option from ResEdit's Resource menu. After sizing the alert in the MiniWindow and entering a 'DITL' ID, you create the 'DITL'. Again, you'll use the "Create New Resource" option.

Figure 6–3 shows a typical 'DITL' resource. The 'DITL' (for dialog item list) lists the *items* in an alert or dialog box. The various items, such as buttons and check boxes, appear in the floating palette in Figure 6–3. You create an item by clicking on its picture in the floating palette and then, with the mouse button still held down, dragging the mouse over to the window. Releasing the mouse button places the item in the window. Figure 6–3 shows a 'DITL' with three items: a static text item and two

push buttons. Alerts are meant to be simple; they don't have provisions for working with check boxes and radio buttons. For those items you'll use a dialog box instead of an alert.

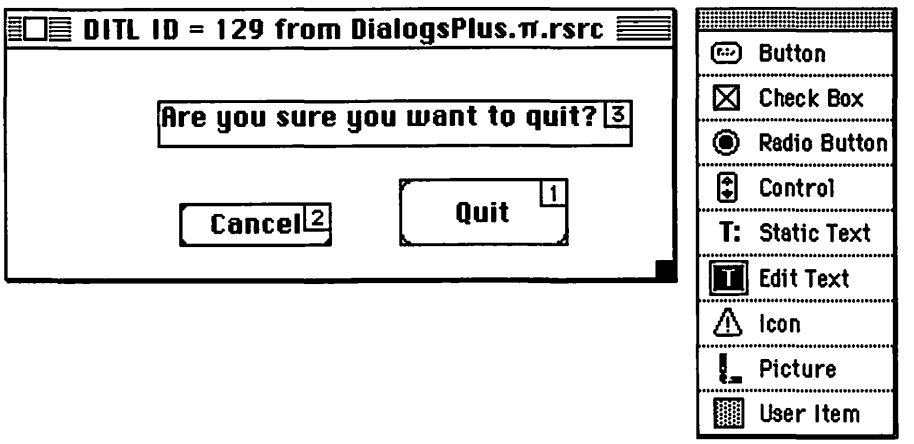


Figure 6–3: *The 'DITL' editor in ResEdit*

To change the name or location of an item, double-click on it. That opens a window that allows you to do just that.

Each item in a 'DITL' has an identifying number. When you select the "Show Item Numbers" option from the 'DITL' menu, ResEdit displays the item number for each item, as shown in Figure 6–3.

In an alert, the button that is item number 1 has special significance. When a program displays an alert that button will appear with an outline, as the Quit button is in Figure 6–1. That tells the user that pressing the keyboard's return key will select that button, just as clicking the mouse button on it would.

ResEdit numbers items in the order you create them. If you aren't satisfied with the numbering of items in a 'DITL' use ResEdit's "Renumber Items" option from the DITL menu to make changes.

With the 'ALRT' and 'DITL' complete you're ready to write the code that brings the alert to the screen.

Alert source code

To load an 'ALRT' resource into memory and display the alert on the screen, use the Toolbox routine *Alert()*. There are two ways to use *Alert()*. The first is for an alert that does not give the user an option, like the alert on the left in Figure 6–4. The second use of *Alert()* is for an alert that presents the user with more than one choice, such as Cancel and Quit. That type of alert is on the right in Figure 6–4.

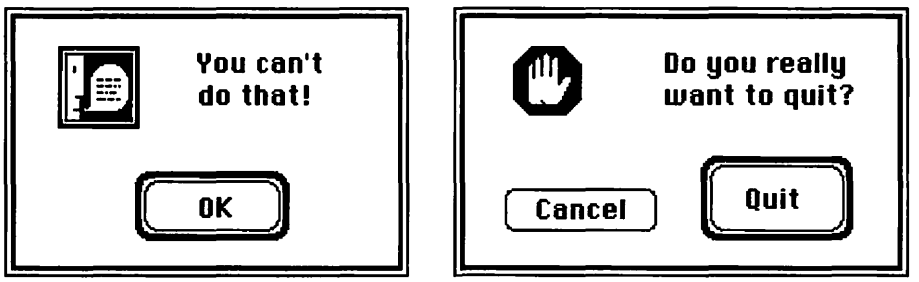


Figure 6–4. *An alert without an option, and one with options*

I show an example of the first usage of *Alert()* in the following code. Simply pass *Alert()* the resource ID of the 'ALRT'.

```
#define     NO_WAY_ALERT     128
#define     NIL               0L

Alert( NO_WAY_ALERT, NIL );
```

By definition, the return type of a call to *Alert()* is of type *short*; that is, a short integer. In the first usage you ignore the *return* type. In the second usage you save the return type. It tells which button the user clicked. Here's an example:

```
#define     QUIT_ITEM         1
#define     CANCEL_ITEM       2
#define     NO_WAY_ALERT    128
#define     NIL               0L

short  alert_item;

alert_item = Alert( NO_WAY_ALERT, NIL );
```

```
if ( alert_item == QUIT_ITEM )
   ExitToShell();

[ else go on with code as if nothing happened ]
```

You may have noticed that the two alerts shown in Figure 6–4 have different icons. There are four variations of the *Alert()* routine. The first, *Alert()*, displays no icon. The other three, *NoteAlert()*, *CautionAlert()*, and *StopAlert()*, each display a different icon. Figure 6–5 shows these icons. All four of the *Alert()* routines have the same two parameters.

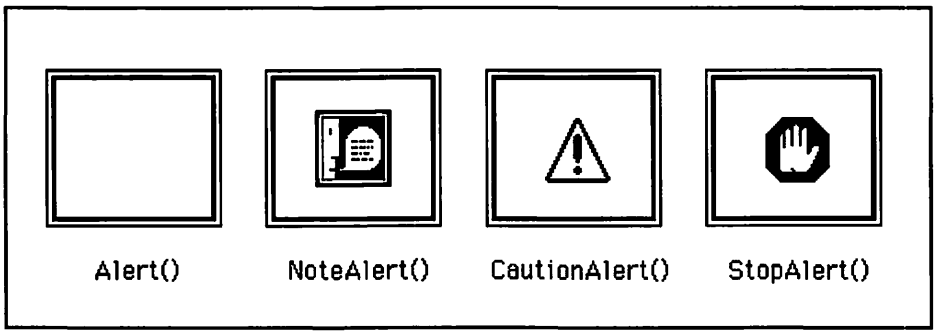


Figure 6–5. *Variations of* Alert() *and the icons they display*

Dialogs

A dialog box is similar to both an alert and a window. A dialog box is like an alert in that it has items in it, but it has a much greater variety of items. A dialog requires a 'DITL' resource, just as an alert does. A dialog box can take the appearance of a window, and, like a window, can be movable. You can think of an alert as a stripped-down dialog, and a dialog as a souped-up window.

Dialogs come in two varieties: modal and modeless. A modal dialog is fixed on the screen—it can't be moved. No action unrelated to the dialog can take place until the dialog is dismissed. A modeless dialog can be moved. Its behavior is similar to a window in that it can contain a title bar that allows the user to drag the dialog. Figure 6–6 shows an example of both types of dialog.

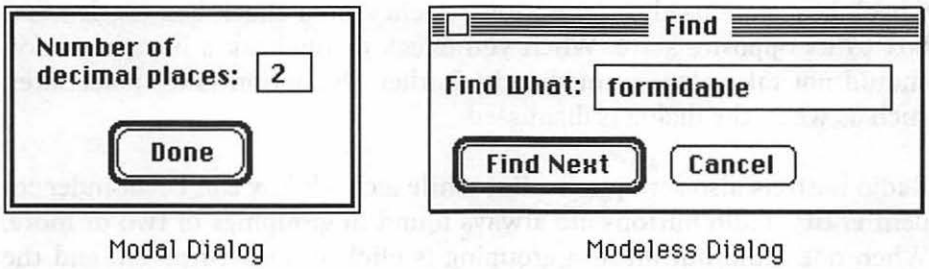


Figure 6–6. *Modal and modeless dialogs*

This chapter covers both types of dialogs, modal and modeless.

Dialog Resources

A dialog can contain several types of items. Figure 6–7 shows the search dialog box from Microsoft Word, a typical dialog with several item types. A brief description of the item types follows.

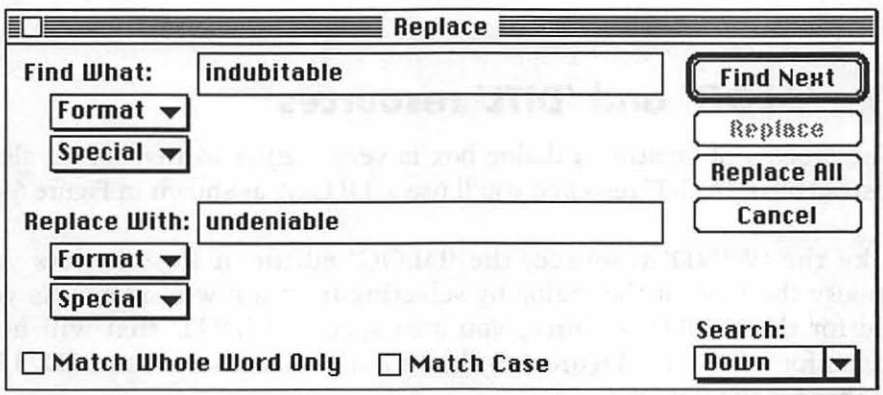


Figure 6–7. *A typical dialog box*

Dialog item types

Almost every dialog contains at least one push button in the form of an OK or Cancel button. When you click on a push button an action will take place immediately, such as the dismissal of the dialog with a click on a button labeled Done.

Check boxes are used to set options. A click on a check box toggles that box to its opposite state. When you check or uncheck a box the action should not take place immediately. Rather, the action takes place later, such as when the dialog is dismissed.

Radio buttons also set options. But while a check box can be an independent entity, radio buttons are always found in groupings of two or more. When one radio button in a grouping is clicked on it turns on, and the button that was previously on is turned off.

Edit text items are your means of supplying text to the computer. Text is typed into the framed rectangle that makes up the item. Dialog text that cannot be edited, such as instructions, is composed of static text items.

The graphics that appear in a dialog can be made up of icons, pictures, or user items. Icons are always 32-by-32 pixels in size. Pictures and user items can be any size. A picture is a 'PICT' resource, while a user item is a free-form type that can be made up of a picture, an icon, or a drawing defined by calls to QuickDraw routines.

The 'DLOG' and 'DITL' resources

The process of creating a dialog box is very similar to that for an alert. Instead of an 'ALRT' resource you'll use a 'DLOG', as shown in Figure 6–8.

Like the 'WIND' resource, the 'DLOG' editor in ResEdit lets you choose the look of the dialog by selecting from a row of icons. As you did for the 'ALRT' resource, you also specify a 'DITL' that will hold items for a 'DLOG'. Figure 6–9 shows a 'DITL' with the same ID, 128, as that for the 'DLOG'.

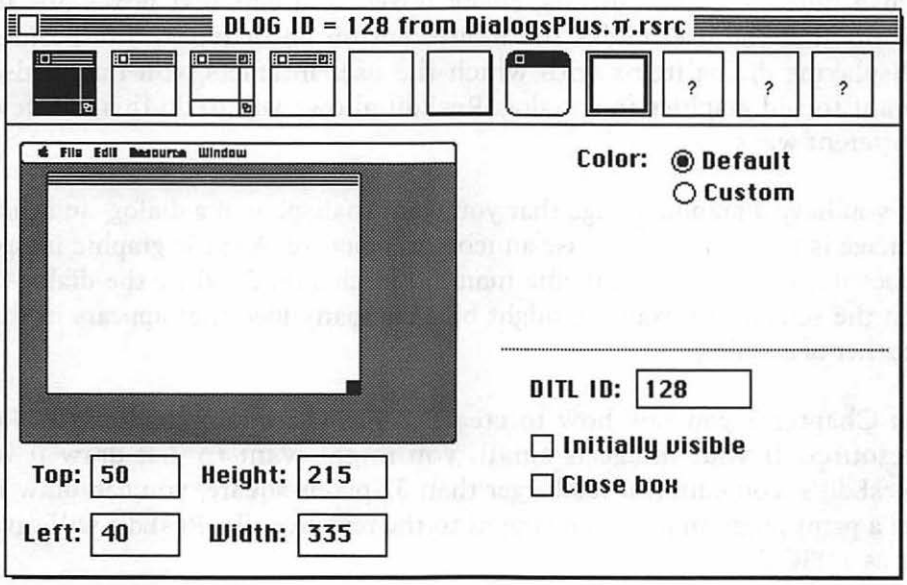


Figure 6–8. *The 'DLOG' editor in ResEdit*

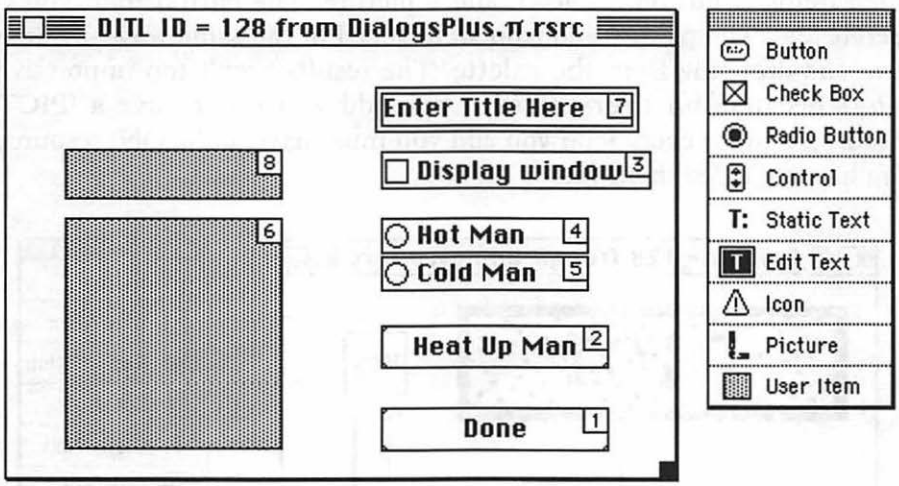


Figure 6–9. *The 'DITL' editor in ResEdit*

When you read about alerts you learned how to add items to the 'DITL' by dragging them from the palette and dropping them into the 'DITL' window. The 'DITL' for a dialog is the same resource type as the 'DITL' for an alert, so you already know how to create the 'DITL' for a dialog.

Push buttons, radio buttons, check boxes, and edit text boxes are all items that the user clicks the mouse button on or types into. Besides displaying dialog items with which the user interacts, you might also want to add graphics to a dialog. ResEdit allows you to do this in a few different ways.

If you have a graphic image that you want to display in a dialog, and that image is to remain static, use an icon or a picture. A static graphic image does not move (as the dancing man did in chapter 3) while the dialog is on the screen. An example might be a company logo that appears in the corner of a dialog.

In Chapter 3 you saw how to create a 'PICT' resource and an 'ICN#' resource. If your image is small, you might want to just draw it in ResEdit's icon editor. If it's bigger than 32-pixels square, you can draw it in a paint program and then copy it to the resource file. ResEdit will save it as a 'PICT'.

If you'd like to include an icon or a picture in a dialog, simply include the proper item in the dialog's 'DITL'. Figure 6–10 shows a 'DITL' with three items: a button, an icon, and a picture. The button item you've seen before. The picture and icon were added in the same way—by dragging and dropping from the palette. The results aren't too impressive. Why? Because for every picture you add you must have a 'PICT' resource, and for every icon you add you must have an 'ICON' resource. You haven't added them yet.

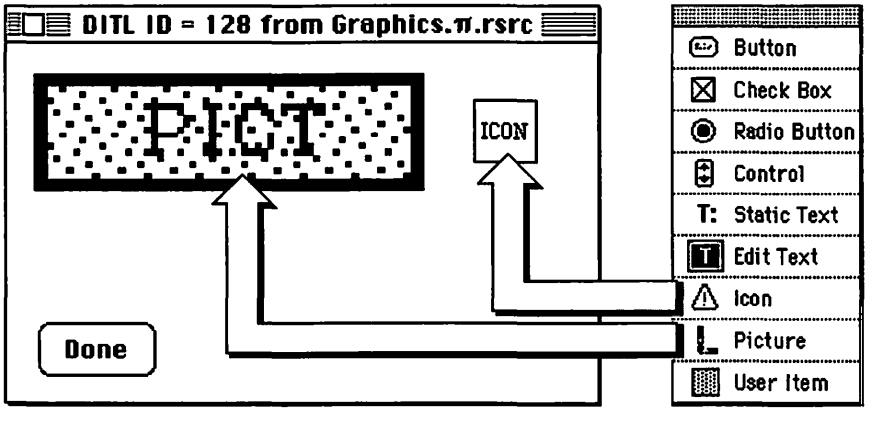


Figure 6–10. *Adding a picture and an icon to a 'DITL'*

The picture item lets you specify the ID of the 'PICT' resource to use. You can double-click on the picture item to edit this information. Figure 6–11 shows that this picture item will be looking for a 'PICT' with an ID of 128.

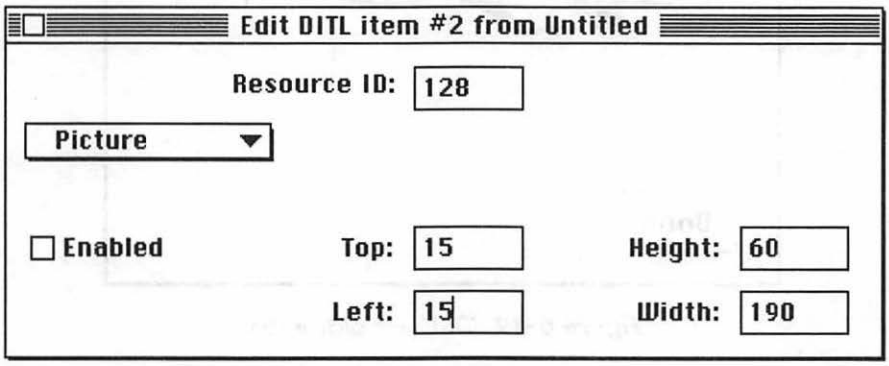


Figure 6–11. *Information window for a picture item*

Notice the check box labeled Enabled in Figure 6–11. When you create certain items like push buttons, check boxes, and radio buttons, this check box will be checked. ResEdit does it for you. If an item is marked as enabled, your program will recognize mouse button clicks on the item. If an item is not enabled, mouse clicks by the user on the item will be ignored. Pictures usually aren't enabled. You can make a picture—or any item—enabled if you want your program to respond to clicks on that item.

From Chapter 3 you know how to create a picture and save it as a 'PICT'. I copied an old standby, one of the four 'PICT's from the dancing man series back in Chapter 3, and pasted it into the resource file. A look at the 'DITL' shows that the rectangle that was the picture item now displays the dancing man picture. Notice that in Figure 6–12 the picture seems distorted. In the window that allows you to enter the 'PICT' ID you can also enter the boundaries for the picture. The 'PICT' you specify will be sized to fit that area. You can double-click on the 'DITL' picture item and change the boundaries any time. The information window was shown in Figure 6–11.

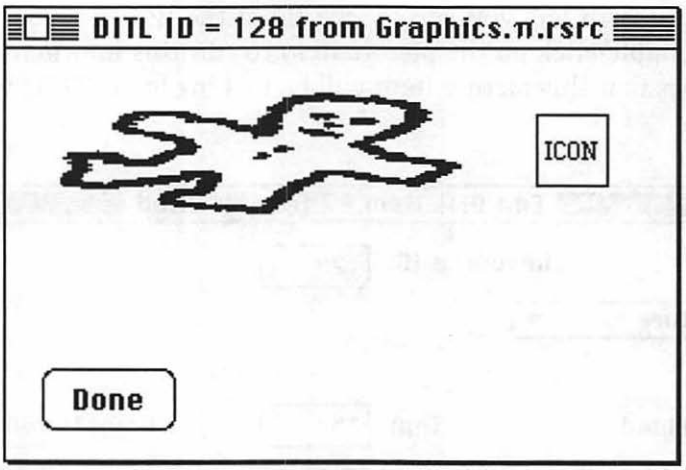


Figure 6–12. *'DITL' with picture item*

Now let's finish off the 'DITL' by making an icon for display in the icon item. Double-click the icon item to set the resource ID of the 'ICON' resource to display, just as you did for the picture. In Chapter 3 you saw how to use the icon editor to create an 'ICN#' resource. You'll use this icon editor again for the 'ICON' resource. The 'ICON' here is for the Acme Fence Company, shown in Figure 6–13.

And just what is the difference between an 'ICN#' and an 'ICON'? I knew you'd ask. The 'ICN#' holds a series of related icons—in Chapter 3 I created an application's icon for display in the Finder. I wanted several versions of it—color, black and white, and smaller icons. So I created an 'ICN#'. The 'ICON' resource holds just a single icon. That's all that's needed here, and that's all the 'DITL' looks for, so that's what I used.

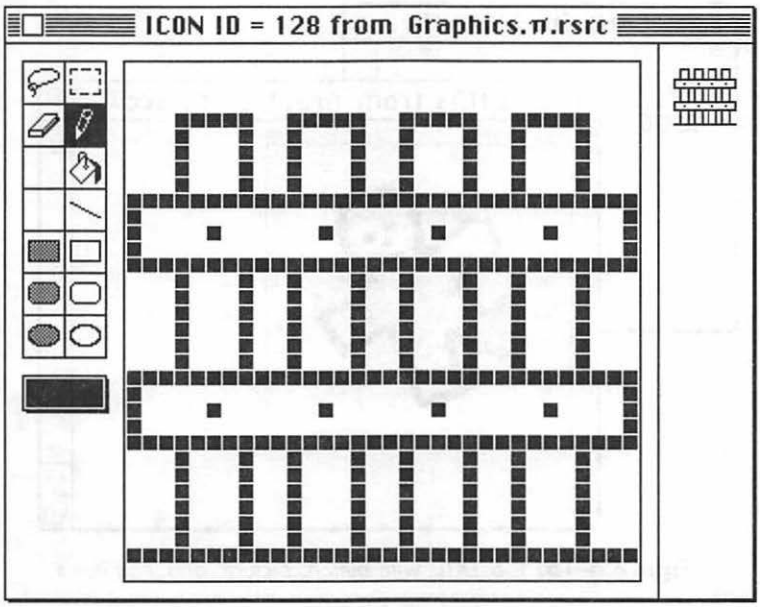


Figure 6–13. *ResEdit's 'ICON' editor in use*

Figure 6–14 shows the 'ICON' and 'PICT' resources. Since the 'ICON' has an ID of 128, it should now appear in the 'DITL', where the icon item has an ID of 128. Figure 6–15 shows that this indeed is the case.

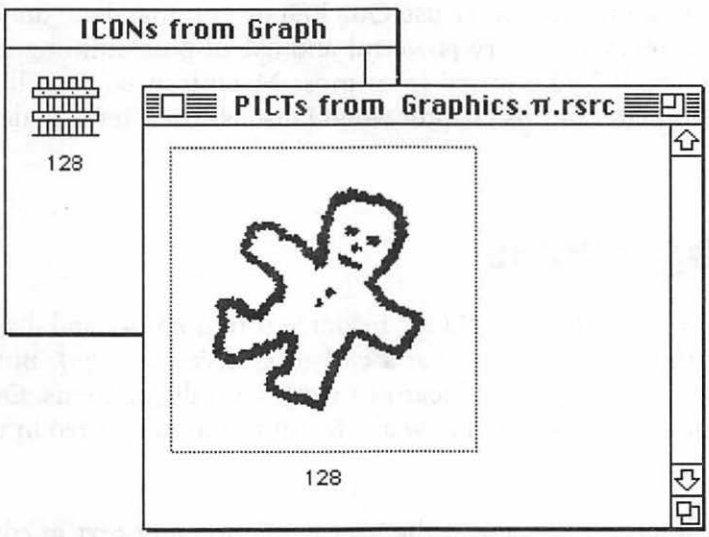


Figure 6–14. *An 'ICON' and a 'PICT' resource in ResEdit*

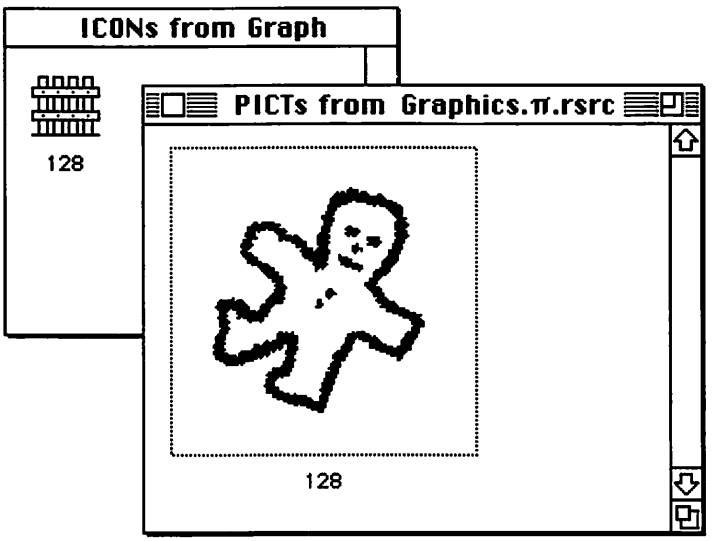


Figure 6–15. *The 'DITL' with button, picture, and icon items*

You've now seen two methods for adding graphics to a 'DITL'. There's a third way: the user item. When you select the user item from the palette and drag and drop it into the 'DITL' window, a mysterious gray box appears. The only feature you can change in a user item is its size—the box will always remain gray. Where do the graphics come from? Your source code will determine that. Through your source code you'll be able to display a picture or icon or use QuickDraw commands to draw something. User items are a very powerful and useful programming tool, and something you'll find omitted from most Macintosh books. I'll remedy that situation later in this chapter when I discuss user items in detail.

Dialog Items

When you want to load a 'DLOG' resource into memory and display the dialog on the screen, you'll use a call to *GetNewDialog()*. But before working with a dialog, you'll learn to work with dialog items. Once you open a dialog you'll need to know all the information covered in the next four sections.

Once a dialog is on the screen the user is free to enter text in edit boxes or click in check boxes, radio buttons and push buttons. It's up to you to

write the code that responds to these user actions. Resources are a great help in designing and implementing a program's interface, but it's still up to you the programmer to write the code that makes things work. You didn't think *everything* would be as easy as creating resources, did you?

Getting dialog item information

Chapter 2 introduced you to handles. You've had occasion to use them since then; for example, when you used a *PicHandle* to draw a 'PICT' resource to a window in Chapter 3. A handle is your program's link to an object in memory—an object that the Memory Manager may shift, or relocate, in memory. Handles play a very important part in dealing with dialogs. When a dialog is on the screen you'll want to examine, and perhaps set, the state of items in it. Before you can work with any dialog item you need to get a handle to it.

GetDItem() is a Toolbox call you'll become very familiar with. You tell *GetDItem()* what dialog you're working with, and which item in that dialog you're interested in. In return, *GetDItem()* gives you three pieces of information about the item: the type of the item, the rectangle that surrounds, or bounds, the item, and a handle to the item. In most instances your only concern will be with the item's handle. Here's a call to *GetDItem()*:

```
#define    DONE_BUTTON_ITEM    1
#define    CHECK_BOX_ITEM      2
#define    EDIT_BOX_ITEM       3

DialogPtr  the_dialog;
short      the_type;
Handle     the_handle;
Rect       the_rect;

GetDItem(the_dialog, EDIT_BOX_ITEM, &the_type, &the_handle, &the_rect);
```

After the above call to *GetDItem()* is complete, your program can use *the_handle* to obtain the text that the user typed in the dialog's edit box or to overwrite the text in the edit box with new text. You'll see just how to do that next.

Working with edit text items

If you want to store the string that the user types in an edit text item, call *GetIText()*—get inserted text. First call *GetDItem()* to get a handle to the item. Use that handle in the call to *GetDItem()*, which will return the text as a *Str255* variable. Here's a fragment that gets the text from an edit box and then writes the string to the upper-left corner of the dialog.

```
#define    EDIT_BOX_ITEM      3

DialogPtr  the_dialog;
short      the_type;
Handle     the_handle;
Rect       the_rect;
Str255     the_string;

GetDItem( the_dialog, EDIT_BOX_ITEM, &the_type, &the_handle, &the_rect );

GetIText( the_handle, the_string );

SetPort( the_dialog );
MoveTo( 15, 15 );
DrawString( the_string );
```

If you want to replace the text that's in an edit box, call *SetIText()*.

```
GetDItem( the_dialog, EDIT_BOX_ITEM, &the_type, &the_handle, &the_rect );
SetIText( the_handle, "\Welcome!" );
```

GetIText() always retrieves an edit text item value as a *Str255* type, even if the user has typed in a number. If you want to convert this string to a number, use *StringToNum()*.

```
short   the_type;
Handle  the_handle;
Rect    the_rect;
Str255  the_string;
long    the_long;

GetDItem( the_dialog, EDIT_BOX_ITEM, &the_type, &the_handle, &the_rect );
```

```
GetIText( the_handle, the_string );

StringToNum( the_string, &the_long );
```

The following example uses all the Toolbox routines just covered. The code retrieves the text from an edit box, converts it to a number, then changes the text in the edit box to a new string, based on the entered number. Here goes:

```
#define    OUT_OF_RANGE_STR    "\pMust be between 0 and 100."
#define    VALID_NUMBER_STR    "\pValid number entered."

DialogPtr  the_dialog;
short      the_type;
Handle     the_handle;
Rect       the_rect;
Str255     the_string;
long       the_long;

GetDItem( the_dialog, EDIT_BOX_ITEM, &the_type, &the_handle, &the_rect );

GetIText( the_handle, the_string );

StringToNum( the_string, &the_long );

if ( ( the_long < 0 ) || ( the_long > 100 ) )
   SetIText( the_handle, OUT_OF_RANGE_STR );
else
   SetIText( the_handle, VALID_NUMBER_STR );
```

Working with check box items

Some dialog items have a state associated with them, such as on or off. The Macintosh gives these two states values: on is considered to have a value of 1, while off is 0. Items that have a value are called control items. Other dialog items such as icons, pictures, and edit text boxes don't have values associated with them.

A check box is a control item. When the user clicks the mouse button on a check box, you call *GetCtlValue()* to get the control value. Whatever its value, zero or one, you set it to its opposite value using *SetCtlValue()*. Here's an example that does just that.

```
#define    CHECK_BOX_ITEM    2
#define    CONTROL_OFF       0
#define    CONTROL_ON        1

DialogPtr  the_dialog;
short      the_type;
Handle     the_handle;
Rect       the_rect;
short      old_value;

GetDItem( the_dialog, CHECK_BOX_ITEM, &the_type, &the_handle, &the_rect );

old_value = GetCtlValue( ( ControlHandle )the_handle );

if ( old_value == CONTROL_ON )
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_OFF);
else
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );
```

Take special notice that both *GetCtlValue()* and *SetCtlValue()* accept only the Macintosh type *ControlHandle* as a parameter; they do not accept a generic *Handle* type. You must always typecast the handle you get from *GetDItem()*, just as done above.

Working with radio button items

If you understood check boxes, you're half way home to working with radio buttons. Check boxes work independently, and you might have just one in a dialog. Radio buttons are dependent on one another, and work in groups—you must have at least two. When the user clicks on one button in a group, the button that was on previous to the click turns off, and the newly clicked button goes on.

Because of this interdependency, you'll want to keep track of the radio button that's currently on. Do this by creating a global variable that holds the dialog item number of the radio button item that's on. When the user clicks on a radio button, you'll turn what is now the old button off and the new button on. Use *SetCtlValue()* to change the radio button values.

Here's an example that starts out with the first radio button of a set of three switched on. The code turns this button off, then turns the second button on.

```
#define     RADIO_1_ITEM       2
#define     RADIO_2_ITEM       3
#define     RADIO_3_ITEM       4
#define     CONTROL_OFF        0
#define     CONTROL_ON         1

short       Old_Button_Num = RADIO_1_ITEM;

DialogPtr   the_dialog;
short       the_type;
Handle      the_handle;
Rect        the_rect;

GetDItem( the_dialog, Old_Button_Num, &the_type, &the_handle, &the_rect );
SetCtlValue ( ( ControlHandle )the_handle, CONTROL_OFF);

GetDItem( the_dialog, RADIO_2_ITEM, &the_type, &the_handle, &the_rect );
SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );

Old_Button_Num = RADIO_2_ITEM;
```

Notice that the last thing the code does is to update the global variable *Old_Button_Num* to hold the dialog item number of the radio button that was just turned on. Next time around, it will be considered the "old" button.

Modal Dialogs

A modal dialog controls the screen, and no action can take place outside the dialog. In certain cases this disadvantage may cause you to use a modeless dialog instead; I discuss them later in this chapter. If you're simply gathering information to use later—perhaps requesting that the user set some preferences for your program—then a modal dialog will do just fine. Because a modal dialog owns the screen, you don't have to worry about the user interacting with other screen elements such as menus, windows, or other dialogs. That makes the source code for handling a modal dialog much less complex than the code you write for a modeless dialog.

The *DialogRecord*

A dialog, modal or modeless, is based on the *DialogRecord* structure. Earlier I said that a dialog had similarities to a window. That was a bit of an understatement. The first member in a *DialogRecord* is a *WindowRecord*, which means that a dialog is actually a window, with a little embellishment. Following the *WindowRecord* member is a member called *items* that is a handle to the items in the dialog. That's pretty much the difference between a window and a dialog: a dialog has items, a window doesn't.

You use a *DialogPtr* to reference a dialog. Because the first member in a *DialogRecord* is a *WindowRecord*, the first thing a *DialogPtr* points to is a *GrafPort*—just as a *WindowPtr* does. Figure 6–16 shows the process. This set up allows you to use a *DialogPtr* as a parameter to Toolbox calls that require a *WindowPtr* or *GrafPtr*. This can be a confusing point. If you're satisfied that this works, skip the technical note that follows.

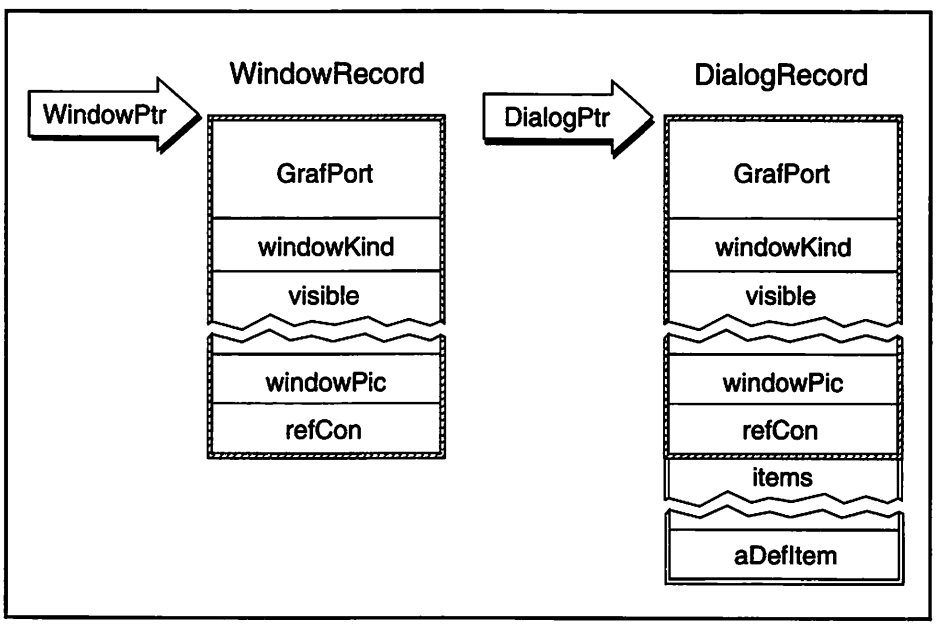


Figure 6–16. *A* DialogPtr *and* WindowPtr *both point to a* GrafPort

Doesn't the idea of being able to use a *GrafPtr*, a *WindowPtr*, and a *DialogPtr* interchangeably almost seem like cheating? If you look at the type definitions of each, you'll see why this works:

```
typedef  GrafPort   *GrafPtr;
typedef  GrafPtr    WindowPtr;
typedef  WindowPtr  DialogPtr;
```

All three types are really pointers to a *GrafPort*. Their names are different as a convenience to programmers.

A *WindowRecord* contains a *GrafPort* and other members. A *DialogRecord* contains a *WindowRecord* and other members. It therefore seems as if a *GrafPort*, *WindowRecord*, and *DialogRecord* should be different sizes. They are. But the first member of each type is the *GrafPort*, so that's what each pointer points to.

You could conceivably have the Window Manager load a window and return either a *GrafPtr* or a *WindowPtr* to your program, as I do below. The Window Manager reserves the same amount of memory for each—the size of the *WindowRecord*, not the smaller size of a *GrafPort*.

```
WindowPtr  the_window_1;
GrafPtr    the_window_2;
Ptr        wind_storage;

wind_storage = NewPtr( sizeof ( WindowRecord ) );
the_window_1 = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
the_window_2 = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
```

Lesson 6-1: The *DialogPtr*

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Modal dialog source code

You load a 'DLOG' resource with a call to *GetNewDialog()*. Then call *ShowWindow()* to display it.

```
DialogPtr  the_dialog;

the_dialog = GetNewDialog( DIALOG_ID, NIL, IN_FRONT );
```

Since a modal dialog controls the screen, you know that it will be dismissed before the program continues. Whatever memory it occupies while it exists will soon be returned to the pool of free memory. Thus, there is no need to reserve your own memory—let it land in memory wherever the Memory Manager puts it. It can't cause fragmentation, because it won't be around to block things.

After creating the dialog, you enter a loop. The loop repeats itself until the user dismisses the dialog; that's how the modal dialog controls the screen. At the heart of the loop is a call to the Toolbox function *ModalDialog()*.

The powerful *ModalDialog()* routine takes control and determines if a mouse click by the user occurs on an enabled item in the dialog. If an enabled item is clicked on, *ModalDialog()* returns the resource item number of the item to your program. Run that number through a switch statement to process the mouse click; that is, base your handling of the mouse click on the item the user clicked on. Figure 6–17 shows this journey from the click of the mouse to *ModalDialog()*.

A modal dialog remains on screen as long as the *ModalDialog()* loop is executing. The loop ends when the loop test condition fails. The usual time for this is when the user clicks the dialog's Cancel, OK, or Done button, as shown in the *Handle_Modal_Dialog()* routine below.

```
#define    DIALOG_ID          128
#define    DONE_BUTTON_ITEM     1
#define    CHECK_BOX_ITEM       2

Handle_Modal_Dialog()
{
   DialogPtr  the_dialog;
   short      the_item;
```

```
    Boolean     all_done = FALSE;

    the_dialog = GetNewDialog( DIALOG_ID, NIL, IN_FRONT );
    ShowWindow( the_dialog );

    while ( all_done == FALSE )
    {
       ModalDialog( NIL, &the_item );
       switch ( the_item )
       {
          case DONE_BUTTON_ITEM:
             all_done = TRUE;
             break;
          case CHECK_BOX_ITEM:
             Set_Check_Box( the_dialog, the_item );
             break;
       }
    }

    DisposDialog( the_dialog );
}
```

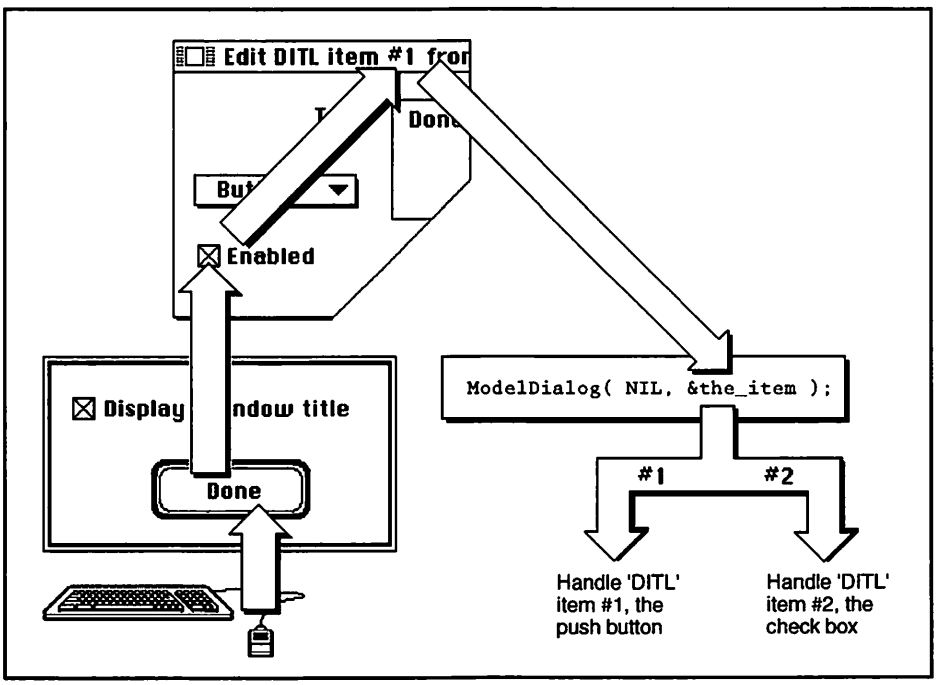
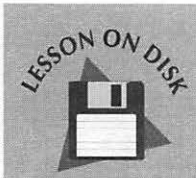


Figure 6–17. *From user action to* ModalDialog()

When the user clicks the Done button, *all_done* is set to true. When the loop again reaches the top, the while test will fail, the loop will end, and the dialog will be dismissed by a call to the Toolbox routine *DisposDialog()*. This chapter's example program makes use of the more powerful modeless dialog. For a working example of a modal dialog see the program presented in Chapter 7.



Lesson 6-2: Using Modal Dialogs

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Modeless Dialogs

To display a modeless dialog on the screen you use the same routine as that for a modal dialog—*GetNewDialog()*. For a modal dialog you didn't specify where in memory the dialog would go, because it wasn't going to be hanging around in memory anyway. For a modeless dialog that might be around for the duration of your program's execution use *NewPtr()* to set the storage. This process is identical to creating a new window. Refer to the previous chapter if you need a review.

```
DialogPtr  the_dialog;
Ptr        dlog_storage;

dlog_storage = NewPtr( sizeof ( DialogRecord ) );
the_dialog = GetNewDialog( DIALOG_ID, dlog_storage, IN_FRONT );
```

Once a modeless dialog is on the screen it needs special handling considerations. In the previous chapter I developed the *Handle_One_Event()* function. This routine is repeatedly called with the purpose of getting an event. Here's that routine, slightly modified.

```
void  Handle_One_Event( void )
{
   Boolean  event_was_dialog;
```

```
    if ( Multifinder_Present == TRUE )
       WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
    else
    {
       SystemTask();
       GetNextEvent( everyEvent, &The_Event );
    }

    event_was_dialog = Handle_Dialog_Event();

    if ( event_was_dialog == FALSE )
    {
       switch ( The_Event.what )
       {
          case mouseDown:
             Handle_Mouse_Down();
             break;
          case updateEvt:
             Handle_Update();
             break;
       }
    }
}
```

This version of *Handle_One_Event()* makes use of a *Boolean* variable called *event_was_dialog*. If a modeless dialog is on the screen, *Handle_One_Event()* wants to know about it. The Toolbox provides a few routines specifically designed to handle an event that takes place in a dialog. I use them in the *Handle_Dialog_Event()* routine I'm about to cover.

Handle_Dialog_Event() checks to see if the current event occurred within a dialog. If it did, *Handle_Dialog_Event()* handles it and returns a value of true. If the event was not dialog-related, *Handle_Dialog_Event()* simply returns a value of false.

Back in *Handle_One_Event()*, the Boolean variable *event_was_dialog* takes on the value returned by *Handle_Dialog_Event()*. If the event was related to a dialog, it's been handled, and this pass through *Handle_One_Event()* is complete. There's no need to enter the switch statement used in the past to handle an event.

I've glossed over the workings of *Handle_Dialog_Event()* so that you'd see the overall technique for handling an event in a program that uses a dialog. Now, it's time to closely examine *Handle_Dialog_Event()*.

```
Boolean  Handle_Dialog_Event( void )
{
   Boolean    event_was_dlog = FALSE;
   DialogPtr  the_dialog;
   short      the_item;

   if ( FrontWindow() != NIL )
   {
      if ( IsDialogEvent( &The_Event )  )
      {
         if ( DialogSelect( &The_Event, &the_dialog, &the_item ) )
         {
            switch ( the_item )
            {
               case DONE_BUTTON:
                  All_Done = TRUE;
                  break;

               [ a "case" to handle each enabled item in the dialog ]
            }

            event_was_dlog = TRUE;
         }
      }
   }
   return ( event_was_dlog );
}
```

The first thing *Handle_Dialog_Event()* does is call the Toolbox routine *FrontWindow()*, which returns a pointer to the frontmost window on the screen. If no windows—or dialogs—are on the screen the routine will return a value of nil. This check verifies that the screen is not empty.

Next, *IsDialogEvent()* is called. This Toolbox routine determines if, at the time of the current event, the frontmost window is a dialog box. If a dialog box isn't in the forefront, you know the event is related to something other than a dialog. So, *IsDialogEvent()* returns a value of false, and *Handle_Dialog_Event()* ends. If the event is a mouse down event in a dialog, *IsDialogEvent()* checks to see if the mouse click occurred in the

dialog's content region. If it didn't, then *IsDialogEvent()* knows the click occurred in the dialog's title bar, and it again returns false. Your program has code for dealing with mouse clicks in a window's title bar, and you can use the same code for a dialog.

DialogSelect() is called next. If execution has made it this far, then it has been established that the event is dialog-related. Now it's time to handle the event. *DialogSelect()* is a powerful routine that will do all of the work for you if a dialog needs updating or activating. In this case, *DialogSelect()* will return a value of false. That tells you the event has been handled, and you're all done.

If the event is dialog related but isn't an update or activate event, *DialogSelect()* doesn't handle it. That's because the event must involve one of the enabled items in the dialog. Each program has different purposes for buttons and check boxes, and *DialogSelect()* has no way of knowing yours. So instead of attempting to handle the event, *DialogSelect()* gives you a pointer to the dialog and the item number of the clicked-on item. It also returns a value of true to signal that processing is now up to you.

At this point handling of the event is dependent on your program's dialog. Use a *switch* statement to determine which item to handle. This *switch* statement is the same one you saw in this chapter's section on working with modal dialogs.

After the *switch* statement, you can consider the event both dialog-related and handled. Now it's safe to set the local *Boolean* variable (initialized to false) to true. Return the value of this variable to the calling routine *Handle_One_Event()*. That lets *Handle_One_Event()* know whether more work needs to be done.

Now, I'll summarize things with Figure 6–18.

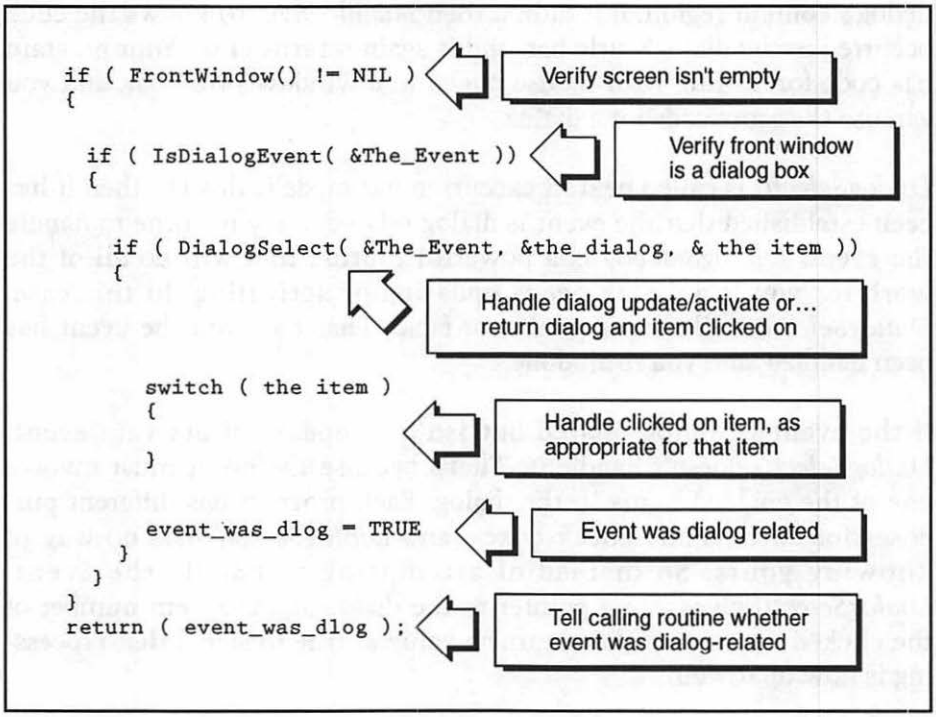


Figure 6–18. *Handling a dialog-related event*

Lesson 6–3: Using Modeless Dialogs

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Using User Items

There may be a time when you want to include an item in a dialog box as your program executes, but not beforehand. That makes placing the item in the 'DITL' impossible. For example, your program might display one of two pictures, depending on the action the user takes. You could include a picture item in your 'DITL', but which 'PICT' ID would you

specify? A problem like this can be overcome using a resource type called a user item.

The user item resource

The user item is a dialog item type tailor-made for situations like the above. When you add a user item to a 'DITL' it appears as a gray box. One such item appears on the left of the 'DITL' in Figure 6–19.

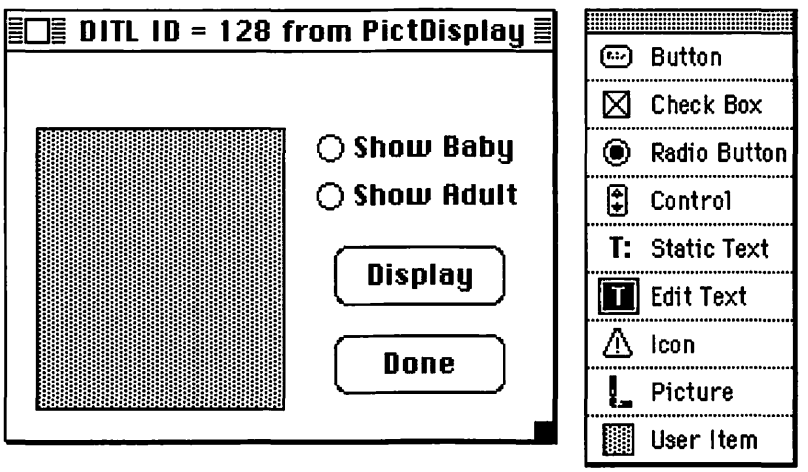


Figure 6–19. *A 'DITL' with a user item*

As with any other resource type, you can double-click on a user item to open the window that allows you to move and resize the item. After that, your job in ResEdit is done. The rest of the work is accomplished in the source code.

The user item source code

Earlier you saw that the Toolbox routine *DialogSelect()* performs the very helpful task of updating a dialog. When a partially obscured dialog is exposed, *DialogSelect()* will redraw buttons, check boxes, icons, and pictures: all the items that are the contents of the dialog box. (Without any work on your part. Very helpful indeed.) The Dialog Manager knows exactly how to redraw these items without any help from you because you defined these items in the 'DITL' resource, just as they are to appear

in the dialog. The Manager uses these definitions when it first displays a dialog and again when it has to update, or refresh, a dialog box.

The Dialog Manager can't update a user item on its own, as it can other item types. The 'DITL' definition of a user item is incomplete; it just shows the display rectangle that will hold the item. Figure 6–20 shows an example of a dialog that uses the 'DITL' from Figure 6–19. An alert is obscuring part of the dialog. I've taken some liberties by showing you what appears under the alert. A real alert would, of course, hide everything behind it. When the alert is dismissed the dialog will need updating.

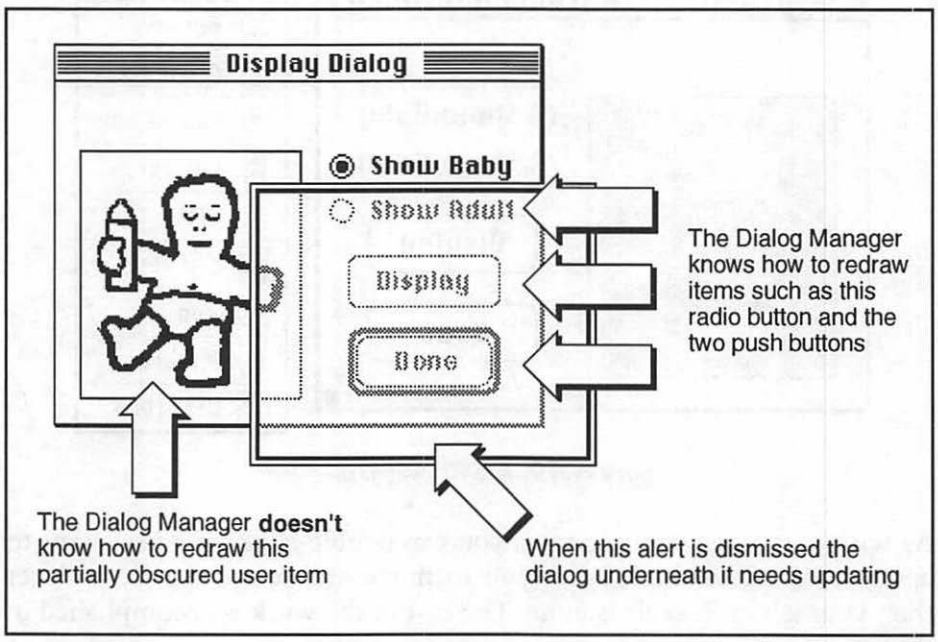


Figure 6–20. *Dialog items will need updating by the dialog manager*

You provide the Dialog Manager the help it needs to update a user item by writing a function that tells the Dialog Manager just what to draw in the display rectangle of the user item. You write this function, but you never call it directly. That's right: your source code never makes a call to the user item routine. Instead, you associate the function with the user item. You bond the two together so that whenever an update occurs the Dialog Manager will call your function, on its own.

You know that *GetDItem()* is used to get information about an item: its type, its display rectangle, or a handle to it. I've been using *GetDItem()* to get a handle to the item. There's a companion routine to *GetDItem()*, called *SetDItem()*, and it's used—as you may have guessed—to set, or change, information about an item.

When you have a user item, you use *SetDItem()* to override the item's handle and replace it with a drawing procedure that tells the Dialog Manager just what to draw in the rectangle making up the user item. Here's an example:

```
#define    USER_ITEM     3

short   the_type;
Handle  the_handle;
Rect    the_rect;

GetDItem( the_dialog, USER_ITEM, &the_type, &the_handle, &the_rect );
SetDItem( the_dialog, USER_ITEM, the_type, (Handle)Do_User_Item, &the_rect );
```

What the above code does is first call *GetDItem()* to get all the information about a user item with the item number 3. It then calls *SetDItem()* to reset everything just as it was, with the exception of the handle to the item. In place of the handle is a function I call *Do_User_Item()*. In the call to *SetDItem()*, omit the parentheses that normally follow a function name. You aren't calling the function here, you're passing the Dialog Manager the address of the function. Just as the name of an array signifies the memory address of the start of the array, so does the name of a function signify the address of the start of the function. Finally, because *SetDItem()* is looking for a handle, you must typecast *Do_User_Item* to a handle.

The user item function doesn't have to be named *Do_User_Item()*. You can give it any name you want, as long as the name used in *SetDItem()* matches the name of the function you write.

Do_User_Item() is a routine you write that defines what the Dialog Manager should draw in the user item. The Dialog Manager will be expecting *Do_User_Item()* to have the following form:

```
pascal void  Do_User_Item( DialogPtr the_dialog, short the_item)
```

User items appeared when Pascal was the native language of the Macintosh. The Dialog Manager is expecting to see a Pascal function here, and you're giving it a C function. Macintosh C provides a *pascal* keyword; use it here and the Dialog Manager will be happy. The *Do_User_Item()* routine can have any name, but it must have two arguments: a *DialogPtr* and a *short*. That's a requirement you must follow.

Keeping Figure 6–20 in mind, let's see what a user item function might look like. From Figure 6–20 you can assume that the user clicks on one of the two radio buttons, then the Display push button. One of two pictures—a baby or an adult—will then be drawn into the rectangle that bounds the user item.

```
#define    BABY_PICT_ID    128
#define    ADULT_PICT_ID   129

short  Current_Pict;

pascal void  Do_User_Item( DialogPtr the_dialog, short the_item )
{
   short      the_type;
   Handle     the_handle;
   Rect       user_rect;
   GrafPtr    old_port;
   PicHandle  pict_handle;

   GetPort( &old_port );
   SetPort( the_dialog );

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &user_rect );

   if ( Current_Pict == BABY_PICT_ID )
      pict_handle = GetPicture( BABY_PICT_ID );
   else
      pict_handle = GetPicture( ADULT_PICT_ID );

   DrawPicture( pict_handle, &user_rect );

   SetPort( old_port );
}
```

Do_User_Item() draws to a dialog, so it first saves whatever port is currently active, then sets the port to the dialog. Next, it calls *GetDItem()* to get the display rectangle, user rect, that bounds the user item.

The program in which *Do_User_Item()* appears has declared a global variable to keep track of which of the two pictures is currently being displayed. The value of this global variable is based on the settings of the two radio buttons and gets set elsewhere in the program. *Do_User_Item()* uses *Current_Pict* to decide which 'PICT' resource to load and draw.

Earlier I said that *SetDItem()* was the device that binds the user item function to the user item itself. You only have to perform this task once, right after opening the dialog in which the user item appears. Here goes:

```
#define    USER_ITEM     3

DialogPtr  the_dialog;
short      the_type;
Handle     the_handle;
Rect       the_rect;
Ptr        dlog_storage;

dlog_storage = NewPtr( sizeof( DialogRecord ) );
the_dialog   = GetNewDialog( DIALOG_ID, dlog_storage, IN_FRONT );

if ( the_dialog == NIL )
   ExitToShell();

GetDItem( the_dialog, USER_ITEM, &the_type, &the_handle, &the_rect );
SetDItem( the_dialog, USER_ITEM, the_type, (Handle)Do_User_Item, &the_rect );

Current_Pict = BABY_PICT_ID;
```

At this point 'DITL' item #3, *USER_ITEM*, is bound to the *Do_User_Item()* function. You set *Current_Pict* to one of the two 'PICT' IDs to start things off. From here on, if the dialog needs updating you're out of the loop; the Dialog Manager knows to call *Do_User_Item()* and take care of things. Your code will never directly make a call to *Do_User_Item()*. Rather amazing, isn't it? You can write a function, then leave it to the Mac to call it when it wants!

I've shown how to handle the case of a single user item in a dialog. What if you want to have more than one? Now that you know what to do for one, working with more than one will be simple. Honest. Figure 6–21 adds a second user item to the 'DITL' I've been using.

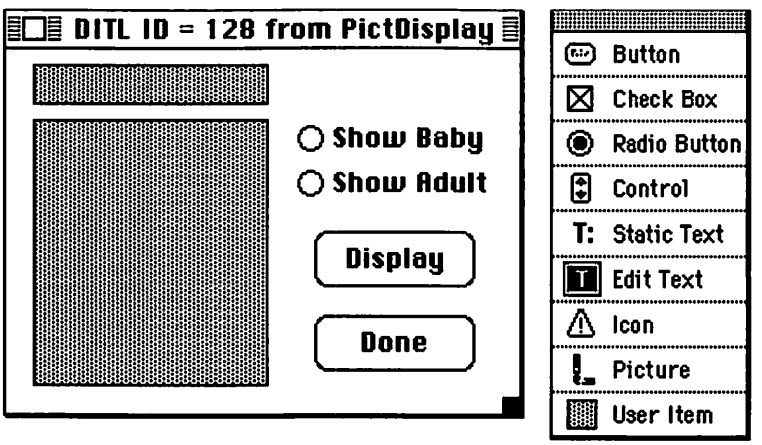


Figure 6–21. *A 'DITL' with two user items*

You know that when a dialog containing a user item is first opened you use *GetDItem()* and *SetDItem()* to associate the user item with the user item function. If you have more than one user item, do this for each item. Now, here's the really neat part. You don't have to write a separate user item function for each item; the same one will do! Below is an example. After that I'll modify the *Do_User_Item()* routine to show how this is possible.

```
#define   USER_ITEM      3
#define   USER_ITEM_2    6

dlog_storage = NewPtr( sizeof( DialogRecord ) );
the_dialog   = GetNewDialog( DIALOG_ID, dlog_storage, IN_FRONT );

if ( the_dialog == NIL )
   ExitToShell();

GetDItem( the_dialog, USER_ITEM, &the_type, &the_handle, &the_rect );
SetDItem( the_dialog, USER_ITEM, the_type, (Handle)Do_User_Item, &the_rect );

GetDItem( the_dialog, USER_ITEM_2, &the_type, &the_handle, &the_rect );
SetDItem( the_dialog, USER_ITEM_2, the_type, (Handle)Do_User_Item, &the_rect );
```

Now, here's a slightly modified Do_User_Item():

```
#define    BABY_PICT_ID    128
#define    ADULT_PICT_ID   129

short  Current_Pict;

pascal void  Do_User_Item( DialogPtr the_dialog, short the_item )
{
   short      the_type;
   Handle     the_handle;
   Rect       user_rect;
   GrafPtr    old_port;
   PicHandle  pict_handle;
   Str255     the_string;

   GetPort( &old_port );
   SetPort( the_dialog );

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &user_rect );

   switch ( the_item )
   {
      case USER_ITEM:
         if ( Current_Pict == BABY_PICT_ID )
            pict_handle = GetPicture( BABY_PICT_ID );
         else
            pict_handle = GetPicture( ADULT_PICT_ID );
         DrawPicture( pict_handle, &user_rect );
         break;

      case USER_ITEM_2:
         if ( Current_Pict == BABY_PICT_ID )
            GetIndString( the_string, STR_LIST_ID, BABY_TITLE_STR );
         else
            GetIndString( the_string, STR_LIST_ID, ADULT_TITLE_STR );
         FillRect( &user_rect, white );
         MoveTo( user_rect.left, user_rect.bottom - 3 );
         DrawString( the_string );
         break;
   }
   SetPort( old_port );
}
```

You may now see how more than one user item can use the same routine. Look at the arguments for *Do_User_Item()*. One is the item number of the item to update. That's the key to the function's power.

When the Dialog Manager calls *Do_User_Item()* it passes along the item number of the user item that needs updating. You use that number when calling *GetDItem()* to get the display rectangle of the user item. You also use the item number in a *switch* statement. In the switch, the code appropriate for this one item is executed. Neat, huh? But there's still more to come.

What if an alert or window is covering the dialog, and, once uncovered, the dialog needs both user items updated? The Dialog Manager will figure this out and will call your *Do_User_Item()* routine twice. On the first call it will pass the item number of one of the user items, and on the second call it will pass the remaining item number. If you weren't amazed before, you've got to be now!

If you could see each line of code executed during this updating you'd see that the *Do_User_Item()* routine gets called twice. If you have a debugger (and if you're using THINK C or Symantec C++, you do) you can test this out on the source code that appears as this chapter's example program. The source code is included on the disk that accompanied this book.

User items are considered mysterious entities by many programmers new to the Macintosh and by many who aren't. And because of the way the Dialog Manager gets involved with your code, user items really are a little mystical. However, as you can see from the above examples, when it comes to writing the code to handle them, they aren't all that tricky.

Color Dialogs

Back in Chapter 4 you saw that creating a color window involved calling *GetNewCWindow()* instead of *GetNewWindow()*. That allowed your window to properly display color graphics drawn with QuickDraw commands. Additionally, you could modify the window's 'WIND' resource so that parts of the window itself, such as the frame or content, contained color. When you modified the 'WIND' in this way ResEdit added a 'wctb' resource to your resource file—a window color table.

For a color dialog, there's no such pair of "GetNew" calls like there is for a window. Instead, you call *GetNewDialog()* whether you want a monochrome or color dialog. What does distinguish one from the other is determined in the resource file.

Analogous to the 'wctb' for windows is the 'dctb' for dialogs—the dialog color table. It gets created when you add color features to a 'DLOG'. In ResEdit, adding color to the elements of a 'DLOG' works just same as for a 'WIND'—refer back to Chapter 4 if you need a refresher.

Now, how does *GetNewDialog()* know whether it should open a monochrome or color dialog? When it goes to load the 'DLOG' resource it looks to see if there's a 'dctb' resource associated with the 'DLOG'. If there is, it creates the new dialog record using a color graphics port. If there is no color table resource for the 'DLOG', a standard monochrome dialog will be created.

Chapter Program: *DialogPlus*

DialogPlus is this chapter's example program. When you run the program you'll come face to face with a modeless dialog box and the inescapable dancing man. The dialog contains an edit text box, a check box, two radio buttons, and two push buttons. By no coincidence, this program demonstrates the use of every item with which a user can interact. A screen shot of *DialogPlus* in action is shown in Figure 6–22.

Clicking on a radio button lets the program know which of two pictures it should display when the Heat Up Man push button is pressed. As a bonus, a click of a radio button also changes the title displayed in the Heat Up Man button—a simple little trick that never fails to amaze onlookers.

Clicking on the Heat Up Man push button causes several things to happen. The program retrieves the user-entered text from the edit text box and displays it in the smaller of the two user items. The program also displays the proper picture in the second user item.

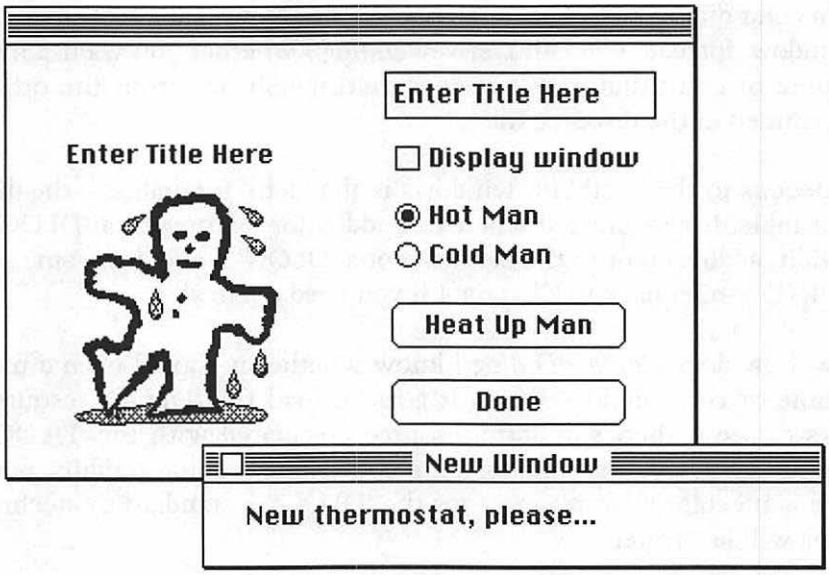


Figure 6–22. *A look at the* DialogPlus *program*

If you click on the Heat Up Man push button, and if the check box is checked, a window will open. The window serves two purposes. First, it gives you a chance to look at the code for a program that contains both a dialog and a window on the screen at the same time—a very real-world kind of thing. Second, you can move the window on and off the dialog to force the Dialog Manager to update things in the dialog, including the two user items: the dancing man and the title that appears above him.

To give a useful example of using an alert, DialogPlus throws up a stop alert when the Done push button is clicked on. This alert gives the user the option of canceling and returning to the program or quitting.

Program resources: *DialogPlus.π.rsrc*

DialogPlus contains a couple of resources you've seen before, the 'WIND' and the 'PICT', and a few that you're seeing for the first time in an example program: an 'ALRT', a 'DLOG', and two 'DITL's.

You're familiar with the 'WIND' resource type, so I don't show it. You've also seen plenty of 'PICT's, but since the two here are new, I show them in Figure 6–23.

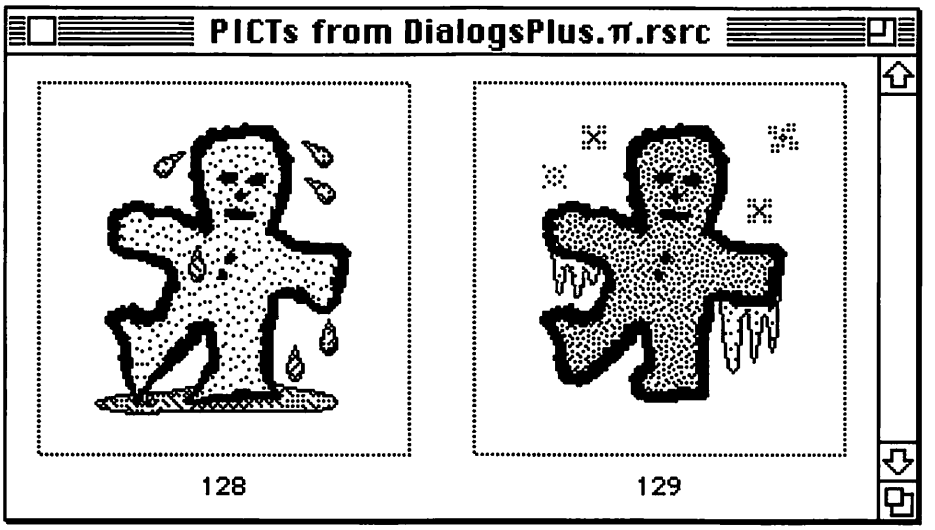


Figure 6–23. *The two 'PICT's from* DialogPlus

DialogPlus uses the alert shown in figures at the start of this chapter. The 'ALRT' has an ID of 129, and so does the 'DITL' associated with it. Figure 6–24 shows the 'DITL'.

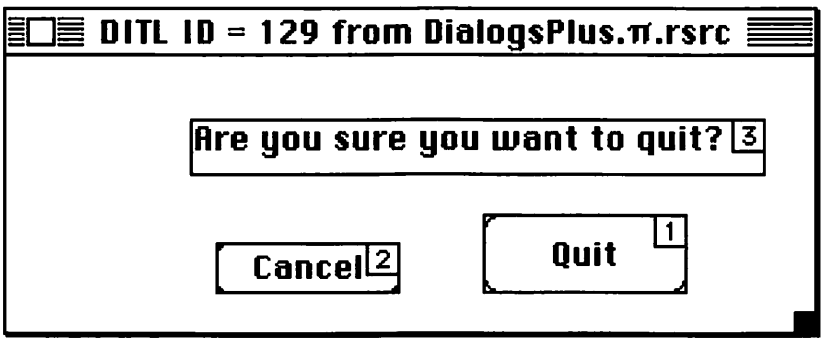


Figure 6–24. *The 'DITL' used by the 'ALRT'*

DialogPlus has one 'DLOG', ID 128, that makes use of 'DITL' 128. Figure 6–25 shows that 'DITL'.

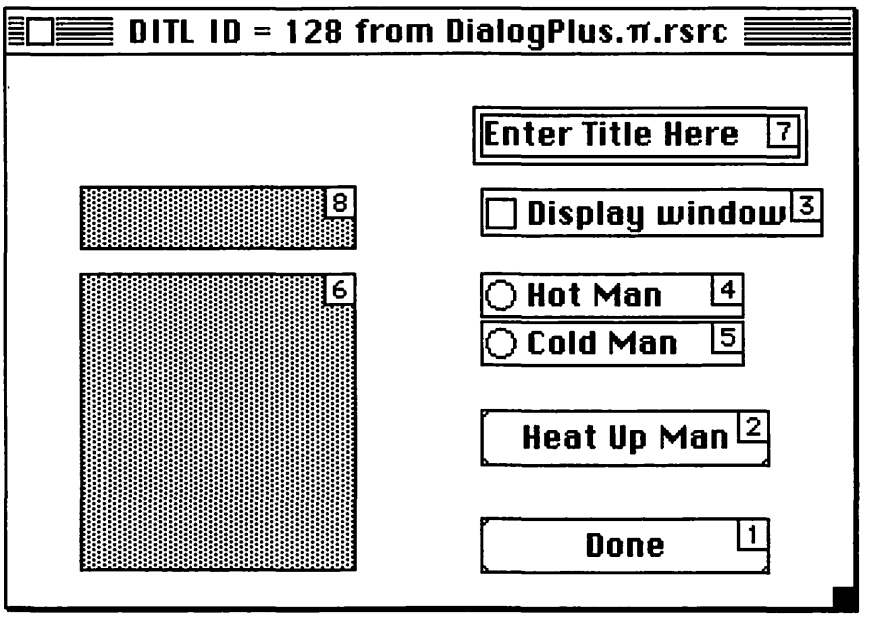


Figure 6–25. *The 'DITL' used by the 'DLOG'*

Both an 'ALRT' and a 'DLOG' use the same type of resource to hold their contents—a 'DITL'. Figure 6–26 shows the relationship of these three types, as used in *DialogPlus*.

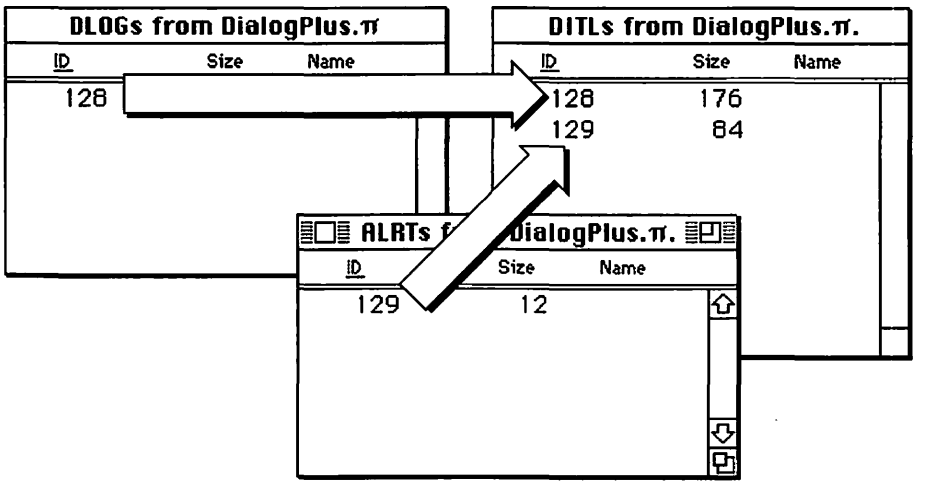


Figure 6–26. *Both an 'ALRT' and a 'DLOG' use a 'DITL'*

Program Listing: *DialogPlus.c*

```
/*++++++++++++++++++++   Include Files   ++++++++++++++++++++*/

#include <Traps.h>


/*++++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void     Initialize_Toolbox( void );
void     Set_Check_Box( DialogPtr, short );
void     Set_Radio_Buttons( DialogPtr , short );
void     Open_Window( void );
void     Open_Dialog( void );
pascal   void   Do_User_Item( DialogPtr, short );
void     Change_Man( DialogPtr, Rect );
void     Draw_Title( DialogPtr, Rect );
void     Handle_One_Event( void );
Boolean  Handle_Dialog_Event( void );
void     Do_Climate_Button( DialogPtr );
void     Set_Climate_Button( DialogPtr );
void     Handle_Mouse_Down( void );
void     Handle_Update( void );
void     Close_Window( WindowPtr );
/*++++++++++++++++++   Define global constants   +++++++++++++++++*/

#define    WIND_ID                      128
#define    QUIT_ALERT                   129
#define    ALERT_QUIT_ITEM                1
#define    DIALOG_ID                    128
#define    DONE_BUTTON                    1
#define    CLIMATE_BUTTON                 2
#define    WIND_CHECKBOX                  3
#define    HOT_RADIO_BUTTON               4
#define    COLD_RADIO_BUTTON              5
#define    MAN_USER_ITEM                  6
#define    EDIT_TEXT_ITEM                 7
#define    TITLE_USER_ITEM                8
#define    MAN_HOT_PICT_ID              128
#define    MAN_COLD_PICT_ID             129
#define    HOT_TITLE          "\pHeat Up Man"
#define    COLD_TITLE       "\pCool Down Man"
#define    CONTROL_ON                     1
#define    CONTROL_OFF                    0
```

```
#define    NIL                              0L
#define    IN_FRONT          (WindowPtr)-1L
#define    REMOVE_EVENTS                     0
#define    SLEEP_TICKS                      0L
#define    MOUSE_REGION                     0L


/*+++++++++++++++++   Define global variables   ++++++++++++++++++*/

Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
DialogPtr    The_Dialog;
WindowPtr    The_Window;
short        Current_Pict;
short        Old_Button_Num;


/*++++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   Initialize_Toolbox();

   The_Window = NIL;
   The_Dialog = NIL;

   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));
   Open_Dialog();

   while ( All_Done == FALSE )
      Handle_One_Event();
}


/*++++++++++++++++++  Initialize the Toolbox  ++++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
```

```
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*+++++++++++++++  Respond to click in a check box  +++++++++++++*/

void  Set_Check_Box( DialogPtr the_dialog, short the_item )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;
   int     old_value;

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &the_rect );

   old_value = GetCtlValue( ( ControlHandle )the_handle );

   if ( old_value == CONTROL_ON )
      SetCtlValue( ( ControlHandle )the_handle, CONTROL_OFF):
   else
      SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );
}


/*++++++++++++  Respond to click in a radio button  +++++++++++++*/

void Set_Radio_Buttons( DialogPtr the_dialog, short new_button_num )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;

   GetDItem( the_dialog, Old_Button_Num, &the_type, &the_handle, &the_rect );
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_OFF );

   GetDItem( the_dialog, new_button_num, &the_type, &the_handle, &the_rect );
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );

   Old_Button_Num = new_button_num ;
}

/*++++++++++++++++++++  Open a single window  ++++++++++++++++++++*/
```

```
void  Open_Window( void )
{
   Ptr     wind_storage;
   Str255  the_str;

   if ( The_Window == NIL )
   {
      wind_storage = NewPtr( sizeof( WindowRecord ) );
      The_Window   = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );

      if ( The_Window == NIL )
         ExitToShell();

      ShowWindow( The_Window );
   }
   else
      SelectWindow( The_Window );
}


/*+++++++++++++++++  Open a modeless dialog box ++++++++++++++++++*/

void  Open_Dialog( void )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;
   Ptr     dlog_storage;

   dlog_storage = NewPtr( sizeof( DialogRecord ) );
   The_Dialog   = GetNewDialog( DIALOG_ID, dlog_storage, IN_FRONT );

   if ( The_Dialog == NIL )
      ExitToShell();

   GetDItem(The_Dialog, MAN_USER_ITEM, &the_type, &the_handle, &the_rect);
   SetDItem(The_Dialog, MAN_USER_ITEM, the_type, (Handle)Do_User_Item,
            &the_rect);

   GetDItem(The_Dialog, TITLE_USER_ITEM, &the_type, &the_handle, &the_rect);
   SetDItem(The_Dialog, TITLE_USER_ITEM, the_type, (Handle)Do_User_Item,
            &the_rect);

   Current_Pict = MAN_HOT_PICT_ID;
```

```
   Old_Button_Num = HOT_RADIO_BUTTON;
   GetDItem(The_Dialog, Old_Button_Num, &the_type, &the_handle, &the_rect);
   SetCtlValue((ControlHandle)the_handle, CONTROL_ON);

   ShowWindow( The_Dialog );
}


/*+++++++++++++++  Define contents of user items +++++++++++++++++*/

pascal void  Do_User_Item( DialogPtr the_dialog, short the_item )
{
   short      the_type;
   Handle     the_handle;
   Rect       user_rect;

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &user_rect );

   switch ( the_item )
   {
      case MAN_USER_ITEM:
         Change_Man( the_dialog, user_rect );
         break;
      case TITLE_USER_ITEM:
         Draw_Title( the_dialog, user_rect );
         break;
   }
}


/*++++++++++++  Draw contents of one of two user items ++++++++++*/

void  Change_Man( DialogPtr the_dialog, Rect user_rect )
{
   GrafPtr    old_port;
   PicHandle  pict_handle;

   GetPort( &old_port );
   SetPort( the_dialog );

   if ( Current_Pict == MAN_HOT_PICT_ID )
      pict_handle = GetPicture( MAN_HOT_PICT_ID );
   else
      pict_handle = GetPicture( MAN_COLD_PICT_ID );
```

```
   DrawPicture( pict_handle, &user_rect );

   SetPort( old_port );
}


/*++++++++++++  Draw contents of two of two user items ++++++++++*/

void  Draw_Title( DialogPtr the_dialog, Rect user_rect )
{
   short    the_type;
   Handle   the_handle;
   Rect     the_rect;
   GrafPtr  old_port;
   Str255   the_string;

   GetPort( &old_port );
   SetPort( the_dialog );

   FillRect( &user_rect, white );

   GetDItem(the_dialog, EDIT_TEXT_ITEM, &the_type, &the_handle, &the_rect);
   GetIText( the_handle, the_string );

   MoveTo( user_rect.left, user_rect.bottom - 3 );
   DrawString( the_string );

   SetPort( old_port );
}


/*++++++++++++++++++++  Handle a single event  ++++++++++++++++++*/

void  Handle_One_Event( void )
{
   Boolean  event_was_dialog;

   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }
```

```
    event_was_dialog = Handle_Dialog_Event();

    if ( event_was_dialog == FALSE )
    {
        switch ( The_Event.what )
        {
            case mouseDown:
                Handle_Mouse_Down();
                break;
            case updateEvt:
                Handle_Update();
                break;
        }
    }
}


/*+++++++++++++++  Handle a dialog-related event  +++++++++++++++*/

Boolean  Handle_Dialog_Event( void )
{
    Boolean    event_was_dlog = FALSE;
    DialogPtr  the_dialog;
    short      the_item;
    short      alert_item;

    if ( FrontWindow() != NIL )
    {
        if ( IsDialogEvent( &The_Event )  )
        {
            if ( DialogSelect( &The_Event, &the_dialog, &the_item ) )
            {
                switch ( the_item )
                {
                    case DONE_BUTTON:
                        alert_item = StopAlert( QUIT_ALERT, NIL );
                        if ( alert_item == ALERT_QUIT_ITEM )
                            All_Done = TRUE;
                        break;

                    case WIND_CHECKBOX:
                        Set_Check_Box( the_dialog, the_item );
                        break;

                    case HOT_RADIO_BUTTON:
```

```
                case COLD_RADIO_BUTTON:
                    Set_Radio_Buttons( the_dialog, the_item );
                    Set_Climate_Button( the_dialog );
                    break;

                case CLIMATE_BUTTON:
                    Do_Climate_Button( the_dialog );
                    break;
            }
            event_was_dlog = TRUE;
        }
    }
  }
  return ( event_was_dlog );
}


/*++++++++++++++  Change the title of a push button  ++++++++++++++*/

void  Set_Climate_Button( DialogPtr the_dialog )
{
  short   the_type;
  Handle  the_handle;
  Rect    the_rect;
  short   cntl_value;

  GetDItem(the_dialog, CLIMATE_BUTTON, &the_type, &the_handle, &the_rect);

  if ( Old_Button_Num == HOT_RADIO_BUTTON )
    SetCTitle( (ControlHandle)the_handle, HOT_TITLE );
  else
    SetCTitle( (ControlHandle)the_handle, COLD_TITLE );
}


/*+++++++++  Handle a click on the Climate push button  +++++++++*/

void  Do_Climate_Button( DialogPtr the_dialog )
{
  short   the_type;
  Handle  the_handle;
  Rect    the_rect;
  short   cntl_value;

  GetDItem(the_dialog, WIND_CHECKBOX, &the_type, &the_handle, &the_rect);
```

```
   cntl_value = GetCtlValue ( ( ControlHandle )the_handle );
   if ( cntl_value == CONTROL_ON )
      Open_Window();

   if ( Old_Button_Num == HOT_RADIO_BUTTON )
      Current_Pict = MAN_HOT_PICT_ID;
   else
      Current_Pict = MAN_COLD_PICT_ID;

   GetDItem(the_dialog, MAN_USER_ITEM, &the_type, &the_handle, &the_rect);
   Change_Man( the_dialog, the_rect );

   GetDItem(the_dialog, TITLE_USER_ITEM, &the_type, &the_handle, &the_rect);
   Draw_Title( the_dialog, the_rect );
}


/*++++++++++  Handle update of a window (not a dialog)  +++++++++*/

void  Handle_Update( void )
{
   WindowPtr  the_window;
   GrafPtr    old_port;

   the_window = ( WindowPtr )The_Event.message;

   GetPort( &old_port );
   SetPort( the_window );

   TextFont( systemFont );
   TextSize( 12 );

   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      MoveTo( 20, 20 );
      DrawString("\pNew thermostat, please...");
   EndUpdate( the_window );

   SetPort( old_port );
}


/*++++++++++  Handle a click of the mouse button  ++++++++++++*/

void  Handle_Mouse_Down( void )
```

```
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         break;

      case inSysWindow:
         SystemClick( &The_Event, the_window );
         break;

      case inDrag:
         DragWindow( the_window, The_Event.where, &screenBits.bounds );
         break;

      case inGoAway:
         if ( TrackGoAway( the_window, The_Event.where ) )
            Close_Window( the_window );
         break;

      case inContent:
         if ( the_window != FrontWindow() )
            SelectWindow( the_window );
         break;
   }
}


/*++++++++++++++++++  Close one window  ++++++++++++++++++*/

void  Close_Window( WindowPtr the_window )
{
   HideWindow( the_window );
   CloseWindow( the_window );
   DisposPtr( ( Ptr )the_window );
   The_Window = NIL;
}
```

Stepping through the code

Let's now walk through the *DialogPlus* code, pausing the longest at information pertinent to this chapter.

The #define directives

DialogPlus can display a window with ID of *WIND_ID*, an alert with ID *QUIT_ALERT*, and the modeless dialog that has a resource ID of *DIALOG_ID*.

The alert has a Quit button that has a 'DITL' item number of *ALERT_QUIT_ITEM*. The dialog has several items, each referred to in the code by a define: *DONE_BUTTON*, *CLIMATE_BUTTON*, *WIND_CHECKBOX*, *HOT_RADIO_BUTTON*, *COLD_RADIO_BUTTON*, *MAN_USER_ITEM*, *EDIT_TEXT_ITEM*, and *TITLE_USER_ITEM*.

The dialog can display one of two 'PICT' resources: *MAN_HOT_PICT_ID* or *MAN_COLD_PICT_ID*. A title, dependent on the picture displayed, will be written in the dialog. It will be either *HOT_TITLE* or *COLD_TITLE*.

When a control such as a radio button is on it has a value of *CONTROL_ON*. When it's off, it has the value of *CONTROL_OFF*.

Last, here are the five constants you're getting quite familiar with: the *GetNewWindow()* parameters *NIL* and *IN_FRONT*; the initialization parameter *REMOVE_EVENTS*; and the *WaitNextEvent()* parameters *SLEEP_TICKS* and *MOUSE_REGION*.

```
#define    WIND_ID                128
#define    QUIT_ALERT             129
#define    ALERT_QUIT_ITEM          1
#define    DIALOG_ID              128
#define    DONE_BUTTON              1
#define    CLIMATE_BUTTON           2
#define    WIND_CHECKBOX            3
#define    HOT_RADIO_BUTTON         4
#define    COLD_RADIO_BUTTON        5
#define    MAN_USER_ITEM            6
```

```
#define    EDIT_TEXT_ITEM                  7
#define    TITLE_USER_ITEM                 8
#define    MAN_HOT_PICT_ID               128
#define    MAN_COLD_PICT_ID              129
#define    HOT_TITLE          "\pHeat Up Man"
#define    COLD_TITLE       "\pCool Down Man"
#define    CONTROL_ON                      1
#define    CONTROL_OFF                     0
#define    NIL                            0L
#define    IN_FRONT           (WindowPtr)-1L
#define    REMOVE_EVENTS                   0
#define    SLEEP_TICKS                    0L
#define    MOUSE_REGION                   0L
```

The global variables

DialogPlus uses *All_Done*, *Multifinder_Present*, and *The_Event* in dealing with events. The program can have only one dialog open, so it simply declares a global pointer, *The_Dialog*, to keep track of it The same is done for the one window that can be displayed; that variable is *The_Window*. *Current_Pict* keeps track of which of two pictures is currently displayed in the dialog. There's always one radio button on, and its item number is held in the variable *Old_Button_Num*.

```
Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
DialogPtr    The_Dialog;
WindowPtr    The_Window;
short        Current_Pict;
short        Old_Button_Num;
```

The main() function

The *main()* function performs the standard Toolbox initializations, then sets the global window and dialog pointers to *NIL* to let the program know that no windows or dialogs are on the screen. After a quick check for MultiFinder, *Open_Dialog()* is called to open a modeless dialog. Then it's on to the main event loop.

```
void  main( void )
{
```

```
    Initialize_Toolbox();

    The_Window = NIL;
    The_Dialog = NIL;

    Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                           NGetTrapAddress(_Unimplemented, ToolTrap));
    Open_Dialog();

    while ( All_Done == FALSE )
       Handle_One_Event();
}
```

Handling check boxes and radio buttons

This chapter showed you how to handle a mouse click in a check box. *DialogPlus* takes the example code shown earlier in this chapter, surrounds it with a pair of braces, and calls it *Set_Check_Box()*. If you pass in the resource 'DITL' item number of the clicked-on item and a pointer to the dialog box it's in, as done here, you can use and reuse this routine in all your programs. Without modification.

```
void  Set_Check_Box( DialogPtr the_dialog, short the_item )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;
   int     old_value;

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &the_rect );

   old_value = GetCtlValue( ( ControlHandle )the_handle );

   if ( old_value == CONTROL_ON )
      SetCtlValue( ( ControlHandle )the_handle, CONTROL_OFF);
   else
      SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );
}
```

Like *Set_Check_Box()*, the radio button routine *Set_Radio_Buttons()* is a rehash of the code fragment shown in this chapter. It too asks for the item number of the clicked-on item and a pointer to the dialog. It then turns off the old button before turning on the newly clicked one.

```
void Set_Radio_Buttons( DialogPtr the_dialog, short new_button_num )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;

   GetDItem( the_dialog, Old_Button_Num, &the_type, &the_handle, &the_rect );
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_OFF );

   GetDItem( the_dialog, new_button_num, &the_type, &the_handle, &the_rect );
   SetCtlValue( ( ControlHandle )the_handle, CONTROL_ON );

   Old_Button_Num = new_button_num ;
}
```

Opening a window and a modeless dialog

DialogPlus allows just one window to be opened. If it's already open, the pointer to it won't be empty, or nil. In that case, *Open_Window()* will simply call *SelectWindow()* to activate it. Otherwise it opens a window with a call to *GetNewWindow()*.

```
void  Open_Window( void )
{
   Ptr     wind_storage;
   Str255  the_str;

   if ( The_Window == NIL )
   {
      wind_storage = NewPtr( sizeof( WindowRecord ) );
      The_Window   = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );

      if ( The_Window == NIL )
         ExitToShell();

      ShowWindow( The_Window );
   }
   else
      SelectWindow( The_Window );
}
```

The dialog that opens in *DialogPlus* has two user items in it. So *Open_Dialog()* calls *SetDItem()* twice to tell the Dialog Manager to be

on the watch for a routine called *Do_User_Item()* when it's to update time.

The dialog's going to display one of two pictures, so *Open_Dialog()* assigns *Current_Pict* the 'PICT' ID of one of them here. You must turn on one of the radio buttons when you open the dialog; the Dialog Manager doesn't do that for you. Since I decided to show the hot picture, that's the radio button I should, and do, turn on.

```
void  Open_Dialog( void )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;
   Ptr     dlog_storage;

   dlog_storage = NewPtr( sizeof( DialogRecord ) );
   The_Dialog   = GetNewDialog( DIALOG_ID, dlog_storage, IN_FRONT );

   if ( The_Dialog == NIL )
      ExitToShell();

   GetDItem(The_Dialog, MAN_USER_ITEM, &the_type, &the_handle, &the_rect);
   SetDItem(The_Dialog, MAN_USER_ITEM, the_type, (Handle)Do_User_Item,
            &the_rect);

   GetDItem(The_Dialog, TITLE_USER_ITEM, &the_type, &the_handle, &the_rect);
   SetDItem(The_Dialog, TITLE_USER_ITEM, the_type, (Handle)Do_User_Item,
            &the_rect);

   Current_Pict = MAN_HOT_PICT_ID;

   Old_Button_Num = HOT_RADIO_BUTTON;
   GetDItem(The_Dialog, Old_Button_Num, &the_type, &the_handle, &the_rect);
   SetCtlValue((ControlHandle)the_handle, CONTROL_ON);

   ShowWindow( The_Dialog );
}
```

Drawing user items

This program's *Do_User_Item()* routine has the same format as the one shown in this chapter. Instead of drawing right here within the function, *Do_User_Item()* calls either *Change_Man()* or *Draw_Title()* to do it.

```
pascal void  Do_User_Item( DialogPtr the_dialog, short the_item )
{
   short      the_type;
   Handle     the_handle;
   Rect       user_rect;

   GetDItem( the_dialog, the_item, &the_type, &the_handle, &user_rect );

   switch ( the_item )
   {
      case MAN_USER_ITEM:
         Change_Man( the_dialog, user_rect );
         break;
      case TITLE_USER_ITEM:
         Draw_Title( the_dialog, user_rect );
         break;
   }
}
```

If the user item displays the picture that needs updating, the program calls *Change_Man()*. This routine looks at *Current_Pict* to see which 'PICT' to load and display. Notice that the user item's display rectangle is passed to *Change_Man()* so that the routine knows where to draw it.

```
void  Change_Man( DialogPtr the_dialog, Rect user_rect )
{
   GrafPtr    old_port;
   PicHandle  pict_handle;

   GetPort( &old_port );
   SetPort( the_dialog );

   if ( Current_Pict == MAN_HOT_PICT_ID )
      pict_handle = GetPicture( MAN_HOT_PICT_ID );
   else
      pict_handle = GetPicture( MAN_COLD_PICT_ID );

   DrawPicture( pict_handle, &user_rect );
```

```
    SetPort( old_port );
}
```

If the smaller of the two user items is to be updated, *Draw_Title()* does the work. Before drawing a title in the user item, *Draw_Title()* makes sure to clear out the old title by calling *FillRect()* to white out the user item. Then *Draw_Title()* gets a handle to the text edit box to use in a call to *GetIText()*. With the contents of the edit box retrieved, its a simple matter to move the graphics pen into the user item box and draw the string.

```
void  Draw_Title( DialogPtr the_dialog, Rect user_rect )
{
    short    the_type;
    Handle   the_handle;
    Rect     the_rect;
    GrafPtr  old_port;
    Str255   the_string;

    GetPort( &old_port );
    SetPort( the_dialog );

    FillRect( &user_rect, white );

    GetDItem(the_dialog, EDIT_TEXT_ITEM, &the_type, &the_handle, &the_rect);
    GetIText( the_handle, the_string );

    MoveTo( user_rect.left, user_rect.bottom - 3 );
    DrawString( the_string );

    SetPort( old_port );
}
```

Event handling

Before processing an event, *Handle_One_Event()* calls *Handle_Dialog_Event()* to give that routine the opportunity to handle it. If *Handle_Dialog_Event()* doesn't handle the event, then the event was one of two things.

1. The event was window-related; handle the event in the switch as done in the past.

2. The event was in the dialog's title bar, not its content region. A dialog's title bar is the same as a window's title bar, so you can again use the old window code found in the switch section to drag the dialog. As I so often say, "A title bar is a title bar...."

```
void  Handle_One_Event( void )
{
   Boolean  event_was_dialog;

   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   event_was_dialog = Handle_Dialog_Event();

   if ( event_was_dialog == FALSE )
   {
      switch ( The_Event.what )
      {
         case mouseDown:
            Handle_Mouse_Down();
            break;
         case updateEvt:
            Handle_Update();
            break;
      }
   }
}
```

Handle_Dialog_Event() only handles an event if it's dialog-related. Because the user items aren't enabled in the 'DITL', the Dialog Manager ignores mouse clicks on them. Figure 6–27 shows how a mouse button click on each dialog item is handled.

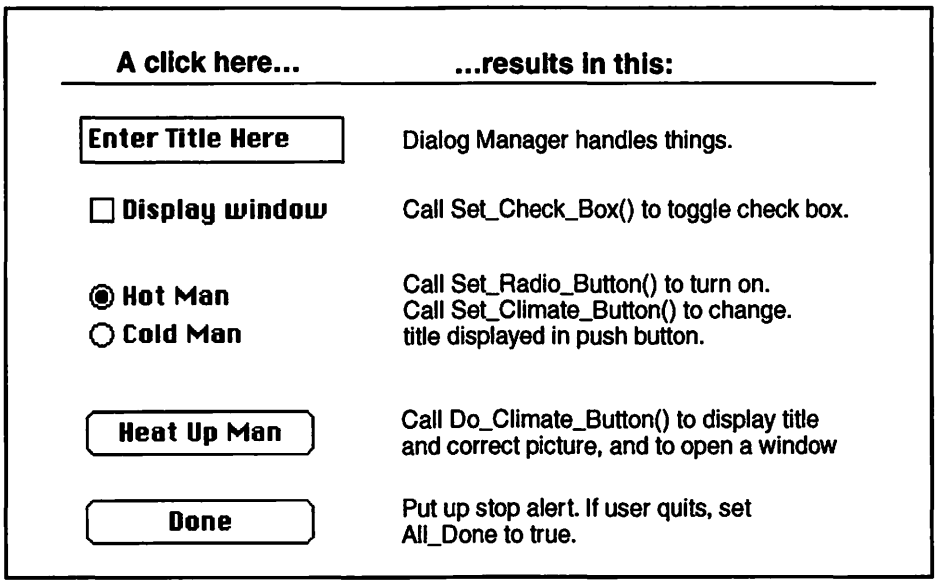


Figure 6–27. *How items are handled*

```
Boolean  Handle_Dialog_Event( void )
{
   Boolean     event_was_dlog = FALSE;
   DialogPtr   the_dialog;
   short       the_item;
   short       alert_item;

   if ( FrontWindow() != NIL )
   {
      if ( IsDialogEvent( &The_Event )  )
      {
         if ( DialogSelect( &The_Event, &the_dialog, &the_item ) )
         {
            switch ( the_item )
            {
               case DONE_BUTTON:
                  alert_item = StopAlert( QUIT_ALERT, NIL );
                  if ( alert_item == ALERT_QUIT_ITEM )
                     All_Done = TRUE;
                  break;

               case WIND_CHECKBOX:
                  Set_Check_Box( the_dialog, the_item );
```

```
                    break;

                case HOT_RADIO_BUTTON:
                case COLD_RADIO_BUTTON:
                    Set_Radio_Buttons( the_dialog, the_item );
                    Set_Climate_Button( the_dialog );
                    break;

                case CLIMATE_BUTTON:
                    Do_Climate_Button( the_dialog );
                    break;
            }
            event_was_dlog = TRUE;
        }
    }
    return ( event_was_dlog );
}
```

A click on a radio button results in a call to *Set_Climate_Button()* to change the title displayed in the push button. This is accomplished by a call to *SetCTitle()*—set control title. I set the button title to match the current radio button setting—*Old_Button_Num*.

```
void  Set_Climate_Button( DialogPtr the_dialog )
{
    short   the_type;
    Handle  the_handle;
    Rect    the_rect;
    short   cntl_value;

    GetDItem(the_dialog, CLIMATE_BUTTON, &the_type, &the_handle, &the_rect);

    if ( Old_Button_Num == HOT_RADIO_BUTTON )
        SetCTitle( (ControlHandle)the_handle, HOT_TITLE );
    else
        SetCTitle( (ControlHandle)the_handle, COLD_TITLE );
}
```

Clicking on the radio button called the Climate button performs a few actions, all taken care of in *Do_Climate_Button()*. First there is a call to *GetDItem()* to get a handle to the check box item. Then a call to *GetCtlValue()* is made to find the value of the check box. Remember that you must typecast the generic handle returned by *GetDItem()* into a

ControlHandle, as done here. If the check box is on, a call to *Open_Window()* results in a display of a small window on the screen.

Next, I determine which radio button is on so that I can display the proper picture. I then set *Current_Pict* to the appropriate 'PICT' ID. To do the actual picture drawing, *Do_Climate_Button()* calls *Change_Man()*, passing along the display rectangle of the user item that will hold the picture. This is the same *Change_Man()* routine called by *Do_User_Item()* during dialog updating.

The routine ends with a call to *Draw_Title()*. This function displays a title in the second user item rectangle.

```
void  Do_Climate_Button( DialogPtr the_dialog )
{
   short   the_type;
   Handle  the_handle;
   Rect    the_rect;
   short   cntl_value;

   GetDItem(the_dialog, WIND_CHECKBOX, &the_type, &the_handle, &the_rect);
   cntl_value = GetCtlValue ( ( ControlHandle )the_handle );
   if ( cntl_value == CONTROL_ON )
      Open_Window();

   if ( Old_Button_Num == HOT_RADIO_BUTTON )
      Current_Pict = MAN_HOT_PICT_ID;
   else
      Current_Pict = MAN_COLD_PICT_ID;

   GetDItem(the_dialog, MAN_USER_ITEM, &the_type, &the_handle, &the_rect);
   Change_Man( the_dialog, the_rect );

   GetDItem(the_dialog, TITLE_USER_ITEM, &the_type, &the_handle, &the_rect);
   Draw_Title( the_dialog, the_rect );
}
```

If *Handle_Dialog_Event()* doesn't handle the current event, it returns a value of false to *Handle_One_Event()*. The *switch* at the bottom of *Handle_One_Event()* is then entered. If it's an update event, and *Handle_Dialog_Event()* didn't handle it, it must be window-related. There's only one window that can be on the screen, so there's no decision-making to perform here; just write to the window.

```
void  Handle_Update( void )
{
   WindowPtr  the_window;
   GrafPtr    old_port;

   the_window = ( WindowPtr )The_Event.message;

   GetPort( &old_port );
   SetPort( the_window );

   TextFont( systemFont );
   TextSize( 12 );

   BeginUpdate( the_window );
      EraseRgn( the_window->visRgn );
      MoveTo( 20, 20 );
      DrawString("\pNew thermostat, please...");
   EndUpdate( the_window );

   SetPort( old_port );
}
```

Handle_Dialog_Event() doesn't handle a mouse down event in a title bar, whether it's a window or dialog. That takes us to *Handle_Mouse_Down()*. This routine, written as it was for windows back in Chapter 5, works here for the window or dialog.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         break;

      case inSysWindow:
         SystemClick( &The_Event, the_window );
         break;

      case inDrag:
```

```
            DragWindow( the_window, The_Event.where, &screenBits.bounds );
            break;

         case inGoAway:
            if ( TrackGoAway( the_window, The_Event.where ) )
               Close_Window( the_window );
            break;

         case inContent:
            if ( the_window != FrontWindow() )
               SelectWindow( the_window );
            break;
      }
   }
```

A click in the window's go away box closes the window. I allocated the memory for the window myself, so I call *CloseWindow()* and *DisposPtr()*. The window's gone, so I set the global pointer *The_Window* to nil so that next time *Open_Window()* is called it will know that a window isn't already open.

```
void  Close_Window( WindowPtr the_window )
{
   HideWindow( the_window );
   CloseWindow( the_window );
   DisposPtr( ( Ptr )the_window );
   The_Window = NIL;
}
```

Chapter Summary

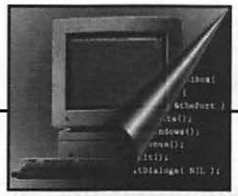
When a user makes a mistake, or is about to do something that could result in a loss of data, a Macintosh program will display an alert. The size and screen location of an alert is defined by an 'ALRT' resource. The items in the alert, such as an informative message and a Cancel or OK button, are defined in a 'DITL' resource. The Toolbox routine *Alert()* displays an alert, using the ID of the passed-in 'ALRT' resource ID.

Dialog boxes can be modal—fixed on the screen; or they can be modeless—movable. The style, size, and screen location of both types are defined by the 'DLOG' resource type. Like an 'ALRT', a 'DLOG' has a related 'DITL' that defines the items that are to appear in the dialog box.

The Toolbox routine *ModalDialog()* does much of the work in handling a modal dialog. It tracks the user's mouse movements and reports back to the program when a user clicks on an item in the dialog.

Modeless dialog boxes require more work on the programmer's part. The Toolbox routine *IsDialogEvent()* determines if a dialog box was the frontmost window when an event occurred. If so, the Toolbox routine *DialogSelect()* is called to handle updates or activates to the dialog. *DialogSelect()* also tracks the user's actions to determine if he clicked on an item in the dialog.

Dialog boxes can contain several types of items. Push buttons, radio buttons, check boxes, and edit text boxes are the most common. The user item is a less used but very powerful item type. This item type allows an item to change as a program executes.



7 Managing Menus

Windows and menus are what originally set Macintosh programs apart from those designed for other computers. Menus allow an application to be nonlinear; that is, it doesn't follow a set sequence of events. Thanks to menus, a program user is free to perform different actions each time he runs a program.

In this chapter you'll learn about the two resource types used to create menus. The 'MENU' is the template for a single menu. The 'MBAR' resource is a collection of 'MENU's used to form a single menu bar.

Here you'll learn that once you get a handle to a menu you can make several changes to the characteristics of a menu and its menu items. You'll disable and enable a menu, change the text and style of text for a menu item, and place a check mark by an item.

About Menus

Every application has its own *menu bar* running along the top of the screen. Your program will define the individual menu names in the

menu bar—the *menus*. It will also define all the *menu items*—the individual items that appear in each menu.

As conditions in a program change, the action of a menu item may not be applicable. At those times you'll want to *disable* that menu item. Disabling the item dims it and makes it impossible to select. Later, when the action of that menu item is usable, you can *enable* it to again make it selectable.

You can use *separator lines* in a menu to logically group menu items. Though technically an item, a separator line is never selectable. It serves only to visually divide a menu into sections.

For a commonly used menu command you can define a *keyboard equivalent*. Rather than making the selection from the menu, the keyboard equivalent allows the user to carry out the menu option by using the command key in conjunction with some other key.

Figure 7–1 shows a menu bar that contains the three standard menus found in almost all Macintosh programs: the Apple, File, and Edit menus.

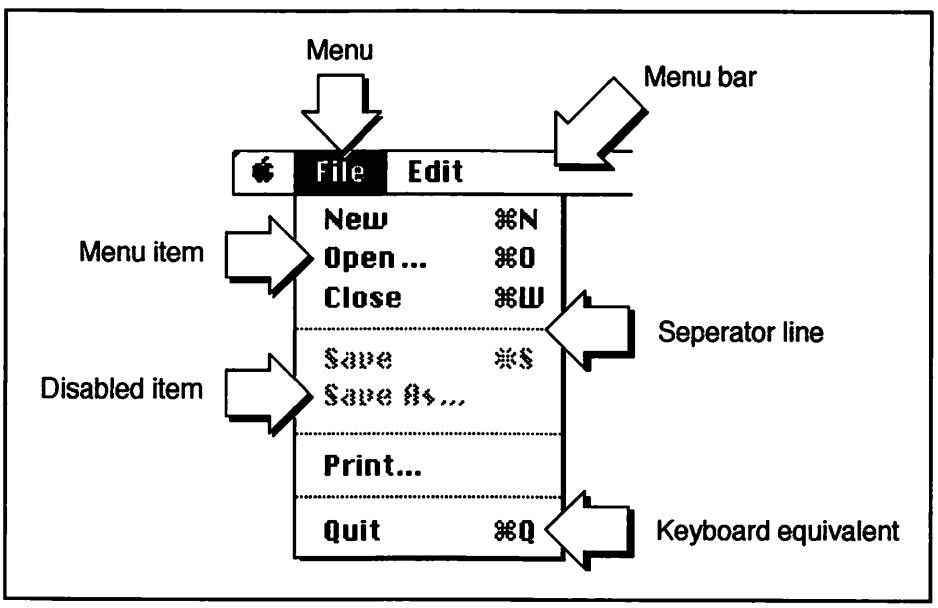


Figure 7–1. *A Macintosh menu*

Menu Resources

You'll rely on two resource types to define the menu and menu items for your program. The 'MENU' resource defines a single menu and the items in it. The 'MBAR' resource groups the individual 'MENU's together into a single menu bar.

The 'MENU' resource

For each menu that your program will have in its menu bar, you'll create a 'MENU' resource. Let's step through that process now.

As is the case for any new resource, select "Create New Resource" from ResEdit's Resource menu. Then double-click on MENU in the Select New Type dialog box. You'll get a window like the one shown in Figure 7–2.

You'll want one of your 'MENU's to represent the Apple menu—the one that holds desk accessories. To do this click, on the menu radio button, as shown in Figure 7–2.

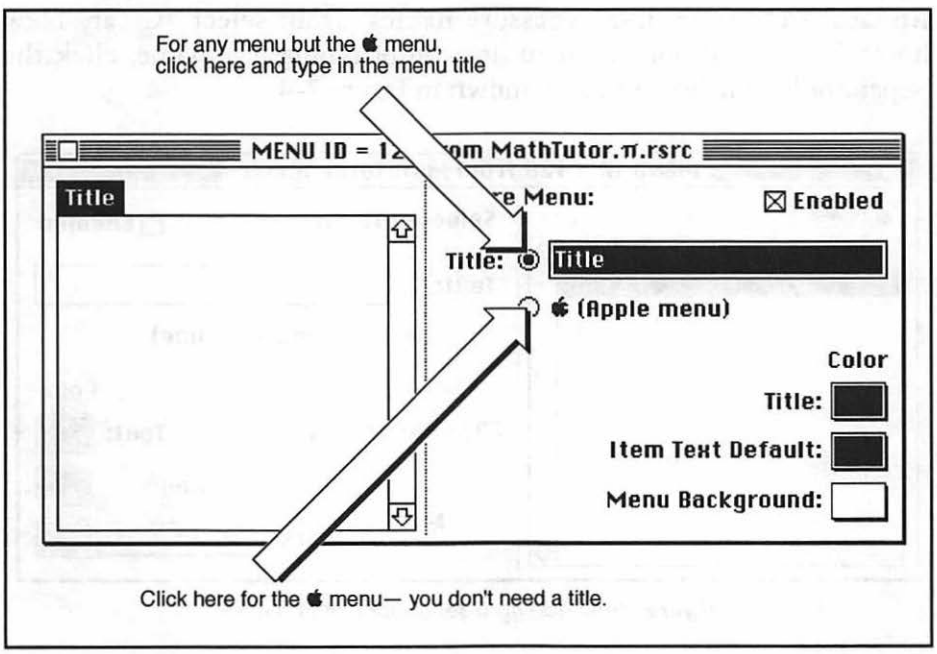


Figure 7–2. *ResEdit's 'MENU' editor*

To add menu items to a 'MENU', whether the Apple menu or another menu, choose "Create New Item" from the Resource menu. Then type the menu item name. Figure 7–3 illustrates typing in the About... item that is typically the first menu item in the Apple menu.

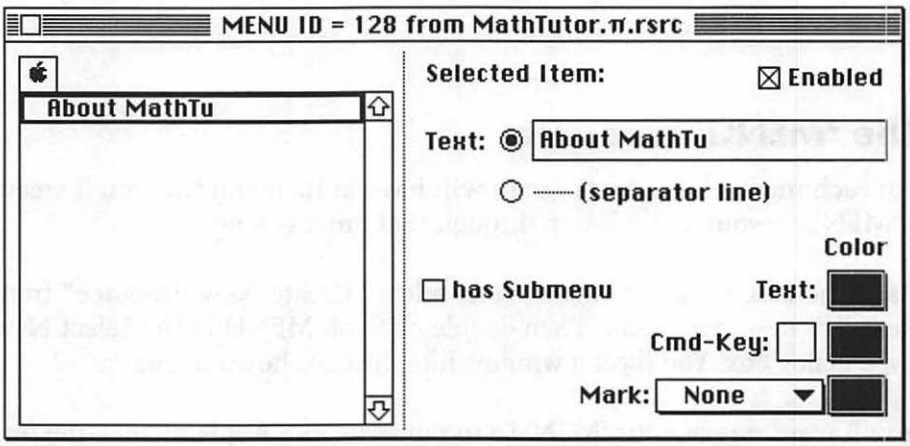


Figure 7–3. *Typing in a menu item name*

To add a separator line between menu items, as is done to separate the About... item from desk accessory names, again select "Create New Item" from the Resource menu. Instead of typing in a name, click the (separator line) radio button, as shown in Figure 7–4.

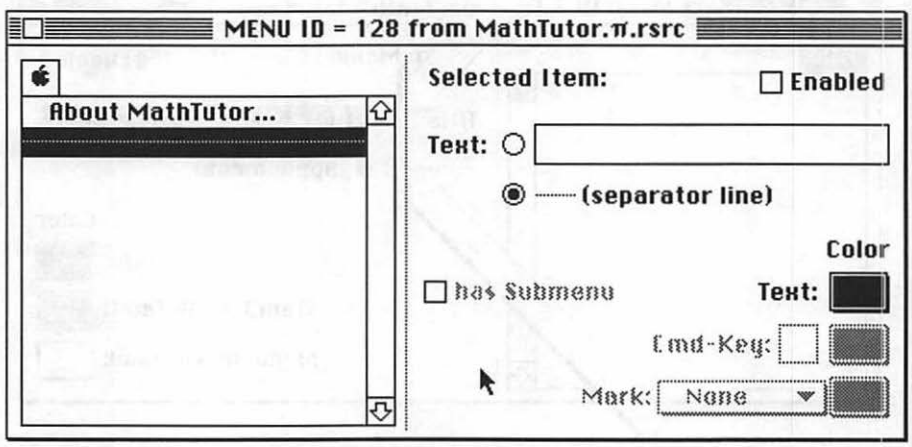


Figure 7–4. *Adding a separator line in a menu*

Figure 7–4 represents a completed 'MENU'. An application's Apple menu contains desk accessories, but you don't add them here because they vary from computer to computer. You'll do that with a Toolbox call when you set up your program's menu bar in the source code.

You need a 'MENU' resource for each menu your program displays. Figure 7–5 shows a second menu—the traditional File menu. Apple recommends that all programs contain the Apple, File, and Edit menus. They're functional, and they give users a sense of familiarity when your program starts up.

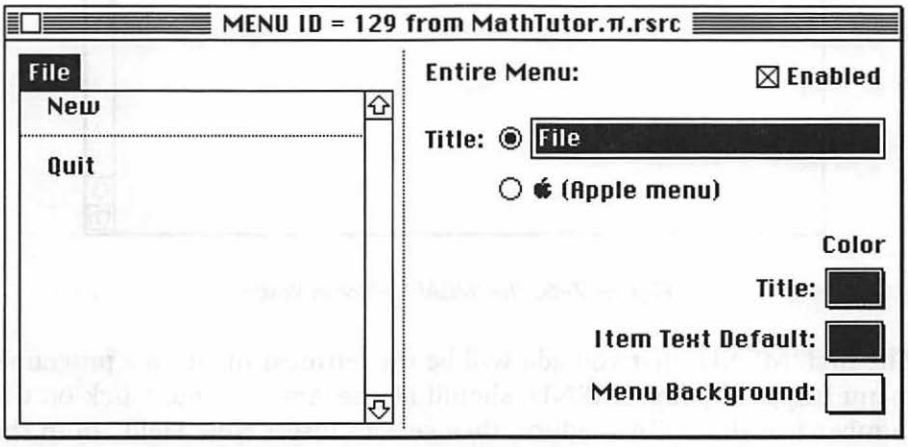


Figure 7–5. *A File menu 'MENU' resource*

The 'MBAR' resource

You've defined each of your program's menus with 'MENU' resources. Now its time to package them together using an 'MBAR' resource. The 'MBAR' defines which 'MENU's will appear in your program's menu bar and in what order.

Why would you have to specify which 'MENU's to use in the menu bar? Why would you define a 'MENU' that *wouldn't* be there? A program can have more than one menu bar. Depending on certain conditions during the running of the program, the menu bar will switch as the program runs.

ResEdit makes creating an 'MBAR' resource easy. Select "Create New Resource" from the Resource menu. You'll see an 'MBAR' editor like that shown in Figure 7–6.

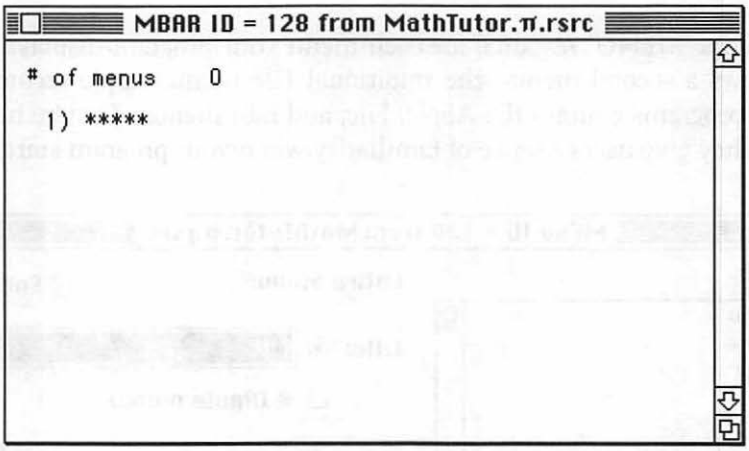


Figure 7–6. *The 'MBAR' editor in ResEdit*

The first 'MENU' that you add will be the leftmost menu in a program's menu bar, so the first 'MENU' should be the Apple menu. Click on the number 1 in the 'MBAR' editor, then select "Insert New Field" from the Resource menu. Enter the ID of the Apple 'MENU' in the box that appears. Since the Apple 'MENU' created earlier had an ID of 128, that's what is entered in Figure 7–7.

After you've entered the resource ID of each 'MENU', the 'MBAR' is complete.

Menu Source Code

The interface between the Menu Manager and you, the programmer, is a particularly good one. There are only a few Toolbox commands you need to become familiar with in order to work with menus.

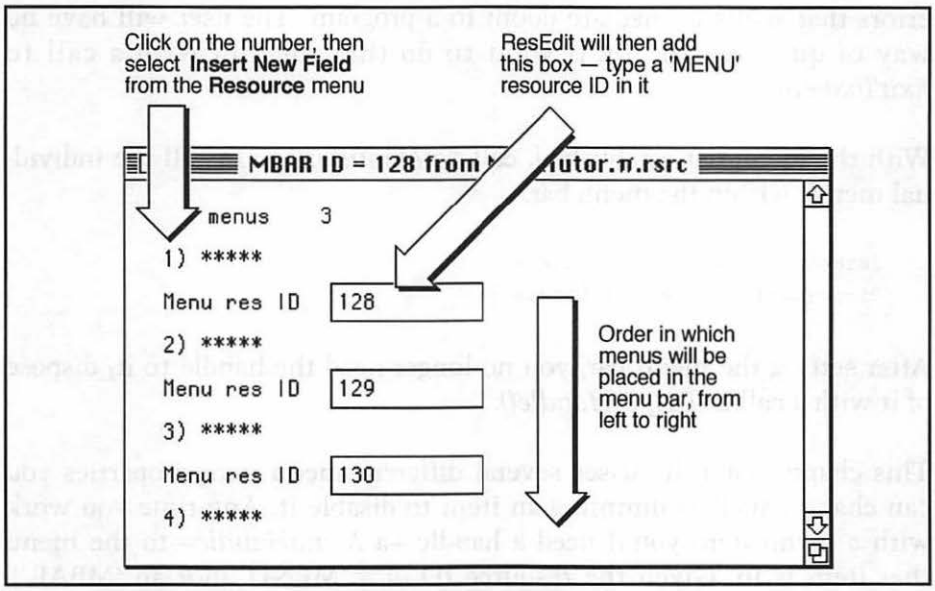


Figure 7-7. *Adding 'MENU's to an 'MBAR'*

Setting up the menu bar

When your program starts up, one of the first things it should do is set up the menu bar; the user will be expecting it to be there immediately. Calling *GetNewMBar()* does this for you. Pass this routine the ID of an 'MBAR' resource and it will create a menu list. The list contains a handle to each individual menu—each 'MENU' resource. Here's a call that uses an 'MBAR' with an ID of 128:

```
#define     MENU_BAR_ID      128

Handle  menu_bar_handle;

menu_bar_handle = GetNewMBar( MENU_BAR_ID );
if ( menu_bar_handle == NIL )
   ExitToShell();
```

It's unlikely that the Menu Manager will fail in its attempt to load your menu resources, but it's a good idea to ensure that the menu bar has been set up. Check the handle returned by *GetNewMBar()* to verify that it's not empty. A missing menu bar, while a rarity, is one of those severe

errors that spells immediate doom to a program. The user will have no way of quitting, so you'll want to do that for him with a call to *ExitToShell()*.

With the menu list established, call *SetMenuBar()* to install the individual menus within the menu bar.

```
SetMenuBar( menu_bar_handle );
DisposHandle( menu_bar_handle );
```

After setting the menu bar, you no longer need the handle to it; dispose of it with a call to *DisposHandle()*.

This chapter later discusses several different menu item properties you can change, such as dimming an item to disable it. Any time you work with a menu item you'll need a handle—a *MenuHandle*—to the menu that item is in. Given the resource ID of a 'MENU' (not an 'MBAR'), *GetMHandle()* returns a *MenuHandle* to that menu.

While you're setting up the menu bar you can get a *MenuHandle* to each individual menu. If you save each as a global variable they'll be available any time your program needs to work with a menu or menu item. Here you get a handle to the Apple menu. *APPLE_MENU_ID* is the resource ID of the 'MENU' that represents the Apple menu.

```
#define      APPLE_MENU_ID      128

MenuHandle   Apple_Menu;

Apple_Menu = GetMHandle( APPLE_MENU_ID );
```

If your program has an Apple menu, and it should, you'll need to make a call to *AddResMenu()*. The contents of the Apple menu vary from computer to computer, so this menu needs some special treatment.

Prior to System 7, desk accessories were stored as resources of type 'DRVR' in the system resource file. On pre-System 7 Macs these resources will have to be placed into the Apple menu. For System 7, desk accessories and anything else the user wants in the Apple menu are stored in the Apple Menu Items folder in the System Folder. The

contents of this folder, collectively called *desktop objects,* will have to be added to the Apple menu.

'DRVR' stands for driver. A driver is the middleman in charge of the transfer of data between a program and a device. A printer is an example of a device.

Whether your program is running on System 7 or an earlier system, a call to *AddResMenu()* will fill the Apple menu.

```
AddResMenu( Apple_Menu, 'DRVR' );
```

With the menus all loaded there's one last thing you must do—display the menu. A call to *DrawMenuBar()* accomplishes this:

```
DrawMenuBar();
```

I've grouped the menu set up calls into one nice neat function, and here it is. I've included three calls to *GetMHandle()*. That's one call for each of the three standard menus included in just about every program: the Apple, File, and Edit menus.

```
#define      MENU_BAR_ID        128
#define      APPLE_MENU_ID      128
#define      FILE_MENU_ID       129
#define      EDIT_MENU_ID       130

MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
MenuHandle   Edit_Menu;

void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      ExitToShell();
```

```
    SetMenuBar( menu_bar_handle );
    DisposHandle( menu_bar_handle );

    Apple_Menu = GetMHandle( APPLE_MENU_ID );
    File_Menu  = GetMHandle( FILE_MENU_ID );
    Edit_Menu  = GetMHandle( EDIT_MENU_ID );

    AddResMenu( Apple_Menu, 'DRVR' );

    DrawMenuBar();
}
```

Lesson 7-1: The Menu Bar

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Handling a click in a menu

A mouse click, whether in a menu bar or not, is an event. It will be captured as a *mouseDown* event in your program's *Handle_One_Event()* function. From there it will be passed on to a routine that handles strictly mouse down events. Here's a refresher:

```
void  Handle_One_Event( void )
{
    [ get event here ]

    switch ( The_Event.what )
    {
        case mouseDown:
            Handle_Mouse_Down();
            break;

        [ handle any other event type ]
    }
}
```

In the past I've used *Handle_Mouse_Down()* to deal with mouse clicks in various parts of a window. *Handle_Mouse_Down()* will still handle all those tasks, but it will additionally take care of a mouse click in the menu bar. Here's the new *Handle_Mouse_Down():*

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;
   long       menu_choice;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         menu_choice = MenuSelect( The_Event.where );
         Handle_Menu_Choice( menu_choice );
         break;

      [ also handle inDrag, inGoAway, etc. ]
   }
}
```

MenuSelect() is the routine that handles menus. This routine will save you a great deal of programming effort. In fact, it's a Toolbox routine so powerful that you'll want to kiss an Apple Toolbox developer for creating it for you! When the user clicks the mouse button in the menu bar, *MenuSelect()* takes control until the user releases the button. Here's a summary of what *MenuSelect()* does:

- It tracks the mouse, dropping down menus as the mouse travels across the menu bar.
- It highlights menu items as the user moves the mouse up and down a dropped down menu.
- It flashes a menu item a few times when the user finally makes a selection.
- It determines the item number and ID of the 'MENU' resource for a menu selection the user makes. It returns this information to your program for processing.

Take a good look at a call to *MenuSelect()*. *MenuSelect()* returns both the ID of the 'MENU' resource that holds the selected menu item and the item number itself. Yet *MenuSelect()* only returns one value—a variable of type *long*. How can this be so?

```
long  menu_choice;

menu_choice = MenuSelect( The_Event.where );
```

MenuSelect() can do this feat by treating the *long* variable as two separate variables. It stores both the 'MENU' ID and the menu item number within the same variable. I'll discuss the simple means to extract these two values in just a bit.

With the display of the menu complete, and the menu selection returned to your program, call a routine to take care of the menu selection. This one is called *Handle_Menu_Choice()*:

```
void  Handle_Menu_Choice( long  menu_choice )
{
   int  the_menu;
   int  the_menu_item;

   if ( menu_choice != 0 )
   {
      the_menu = HiWord( menu_choice );
      the_menu_item = LoWord( menu_choice );

      switch ( the_menu )
      {
         case APPLE_MENU_ID:
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID:
            Handle_File_Choice( the_menu_item );
            break;

         case EDIT_MENU_ID:
            Handle_Edit_Choice( the_menu_item );
            break;
      }
      HiliteMenu( 0 );
```

```
        }
    }
```

If the user scans the menu bar and then backs out of his decision to make a menu selection, *MenuSelect()* will return a value of 0. *Handle_Menu_Choice()* checks to see if this is the case. If not, it's time to extract those two pieces of information tucked inside variable *menu_choice*.

MenuSelect() stores both the 'MENU' ID and the menu item number in one *long* variable. It places the 'MENU' ID in the upper 16 bits of the 32-bit *long* variable and the menu item number in the lower 16 bits. Since the Toolbox performs a little trick like this, it also conveniently provides a couple of routines for extracting the two pieces of information from the one variable: *HiWord()* and *LoWord()*. Figure 7–8 shows this.

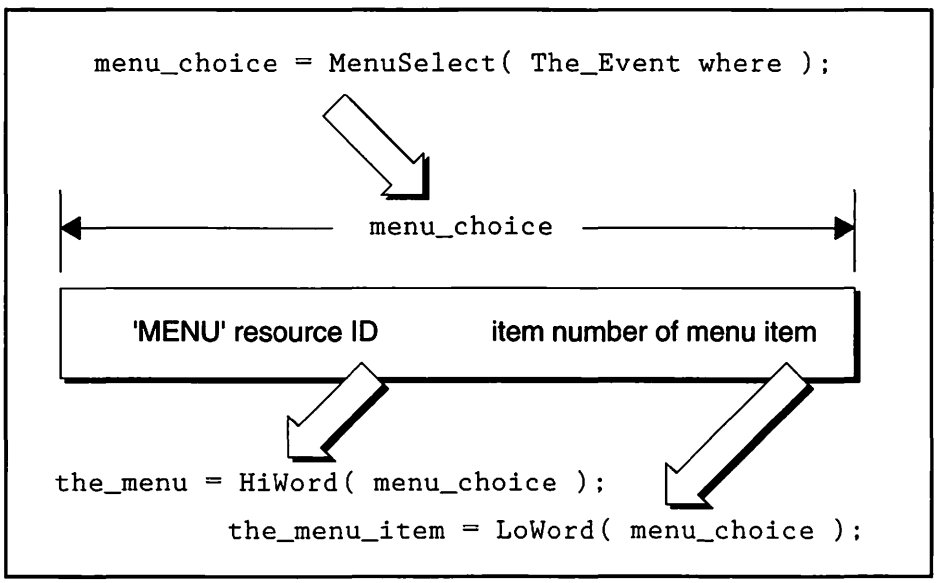


Figure 7–8. *Extracting the menu and menu item from one variable*

Once you know which menu was clicked in, all you need do is branch to a routine written to handle mouse clicks in that particular menu. Of course, pass the item number of the selected menu item. The routine *Handle_Menu_Choice()* works for a program that has just the three standard menus: Apple, File, and Edit.

When a menu item is selected, *MenuSelect()* inverts the menu name in the menu bar. After the menu item selection is handled, your code must call *HiliteMenu()* to again invert the menu name back to its original state.

Handling a click in the Apple menu

How the selection of a particular menu item is handled depends on the item selected. Your program may have a menu item that does things no other program does. But some menu choices are standard fare and are always handled in much the same way. The items in the Apple menu fall into this category.

The first menu item in the Apple menu is usually the About... item. Selecting this item puts up an alert that displays some information about the program's copyright. You learned how to display an alert in the previous chapter.

The remaining items in the Apple menu are desk accessories or, under System 7, any programs or documents the user places in the Apple Menu Items folder. Regardless of what the item is, a call to *OpenDeskAcc()* will get things going. Pass *OpenDeskAcc()* the name of the item to open. You can get the name by calling *GetItem()*. This routine returns the text of a menu item in any menu, not just the Apple menu.

```
#define    ABOUT_ALERT_ID    128

MenuHandle  Apple_Menu;

void  Handle_Apple_Choice( int the_item )
{
   Str255  desk_acc_name;
   int     desk_acc_number;

   switch ( the_item )
   {
      case SHOW_ABOUT_1_ITEM :
         Alert( ABOUT_ALERT_ID, NIL );
         break;

      default :
         GetItem( Apple_Menu, the_item, desk_acc_name );
```

```
            desk_acc_number = OpenDeskAcc( desk_acc_name );
            break;
        }
    }
```

Handling a click in other menus

The format of *Handle_Apple_Choice()* is the format all your menu-handling routines will have. Pass the item number of the selected menu item to the routine, then use a switch statement to get to the code written for that particular item. This chapter's example program gives several options.

This section finishes with a figure that recaps how a click of the mouse gets transformed into a menu selection—Figure 7–9.

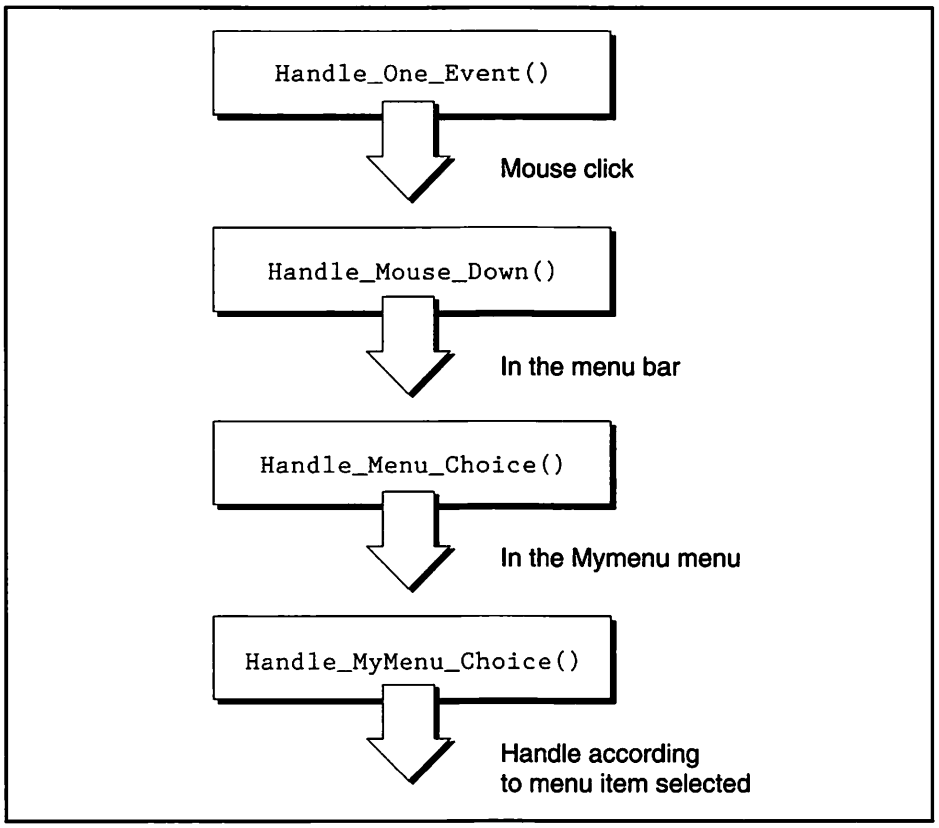


Figure 7–9. *The path from mouse down to menu selection*

Keyboard Equivalents

To make things easy for users you'll want to provide them with *keyboard equivalents* to the most common menu selections. A keyboard equivalent, or keyboard alias, allows the user to bypass the menu bar and make a menu selection from the keyboard. Pressing the Command key in conjunction with one or more other keys does the same thing as using the menu.

Consistency between Macintosh applications is essential to the Apple philosophy of keeping the Mac user-friendly. To this end Apple has reserved some of the keyboard equivalents for common commands found in many Macintosh programs. You can use any of these reserved combinations in your own programs, but you should use them only for the commands shown in Table 7–1.

Table 7–1. *Keyboard equivalents reserved by Apple*

Keyboard Equivalent	Command
⌘-A	Select All
⌘-C	Copy
⌘-N	New
⌘-O	Open...
⌘-P	Print...
⌘-Q	Quit
⌘-S	Save
⌘-V	Paste
⌘-W	Close
⌘-X	Cut
⌘-Z	Undo

You can use ResEdit to add a keyboard equivalent to any menu item. That discussion is next.

The 'MENU' resource

To add a keyboard equivalent to a menu item, use ResEdit to edit the 'MENU' resource in which the menu item appears. Click on the menu item name, then enter the character that will be used along with the Command key. Figure 7–10 shows the addition of a keyboard equivalent to the Quit command in the File menu.

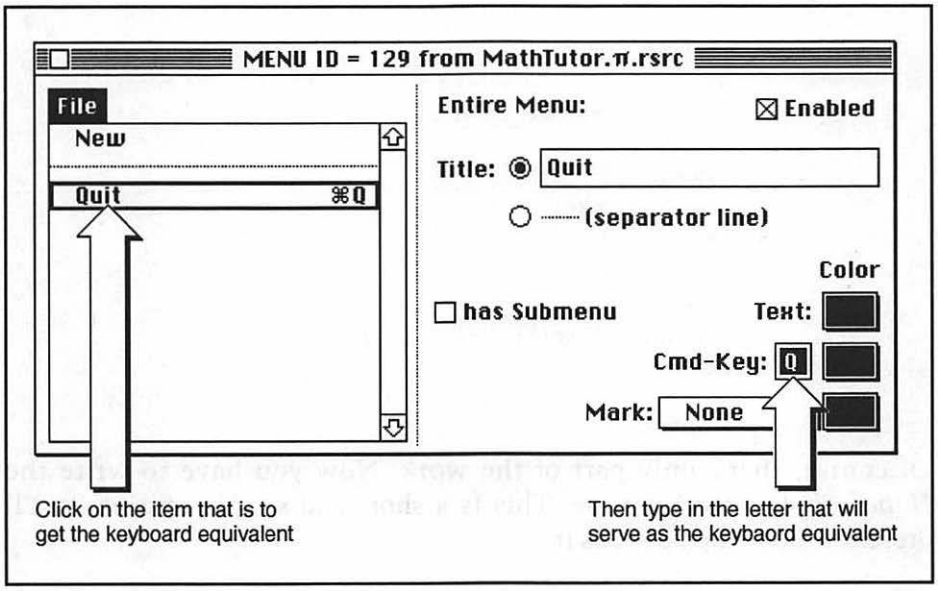


Figure 7–10. *Adding a keyboard equivalent to a 'MENU'*

The character that is typed along with the Command key is displayed in uppercase in a program's menu even though the user won't be using the shift key. Take the Quit keyboard equivalent, Command-Q, for example. The user types the Command key and the letter 'q'. He doesn't use the Shift key to type an uppercase 'Q'.

Handling a keystroke

If you want to include keyboard equivalents in your application, you have to make your program aware of keystrokes. That's something you haven't worried about up to this point. To do this, add a case for a *keyDown* event in your *Handle_One_Event()* routine.

```
void  Handle_One_Event( void )
{
   [ get event here ]

   switch ( The_Event.what )
   {
      case keyDown:
         Handle_Keystroke();
         break;

      case mouseDown:
         Handle_Mouse_Down();
         break;
   }
}
```

Of course, that's only part of the work. Now you have to write the *Handle_Keystroke()* routine. This is a short and simple routine, so I'll present it now, then discuss it.

```
void  Handle_Keystroke( void )
{
   short  chr;
   long   menu_choice;

   chr = The_Event.message & charCodeMask;

   if ( ( The_Event.modifiers & cmdKey ) != 0 )
   {
      if ( The_Event.what != autoKey )
      {
         menu_choice = MenuKey( chr );
         Handle_Menu_Choice( menu_choice );
      }
   }
}
```

When an event involving the keyboard occurs, the *message* element of the event holds the key that was pressed. The *message* field consists of 32 bits that hold more information than just that, though. To access only the portion that contains the character the user typed you'll need to use the constant *charCodeMask* in conjunction with the *bitwise &* operator.

At this point you are only interested in a keystroke performed in conjunction with the Command key. The *modifiers* field of the event holds this information. As you did for the character, though, you have to use the *&* operator on the field to get the information you need. If the result is non-zero, the Command key was down.

One last check: was the key pressed and held down? That's called an auto key, and that's not a keyboard equivalent. If the keystroke survives the battery of tests, then you know that the user held down the command key while pressing a character. That's a keyboard equivalent. At this point call the Toolbox routine *MenuKey()*.

MenuKey() accepts a typed character and returns a *long* integer. The *long* contains both the ID of the menu and the ID of the menu item that the Command key combination represents. With that information you can then call *Handle_Menu_Choice()* to handle things just as if a menu selection had been made.

Hierarchical Menus

To offer the user additional menu choices you can use a hierarchical menu, which is a menu that has a submenu associated with it. Figure 7–11 illustrates an example of a menu with a submenu attached to it.

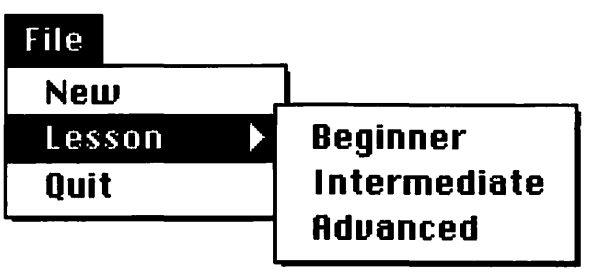


Figure 7–11. *A hierarchical menu*

Adding a hierarchical menu requires a few minimal additions to both your resources and code.

The 'MENU' resource

You designate a submenu for a menu item by checking the "has Submenu" check box in ResEdit. The submenu itself will be a 'MENU' resource. List the resource ID of that 'MENU' in the edit text box labeled ID. Figure 7–12 takes the File 'MENU' resource developed earlier and changes the second item from a separator line to an item named Lesson.

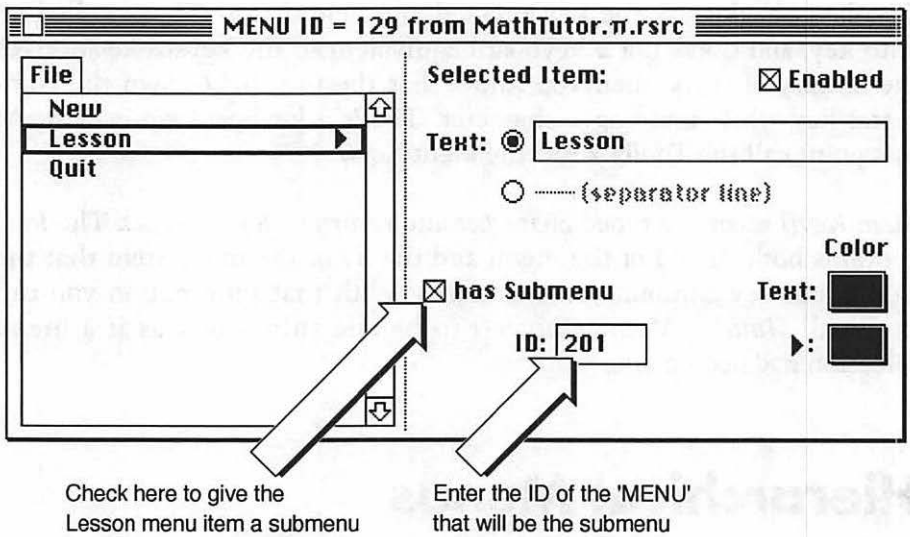


Figure 7–12. *Adding a submenu to a menu item*

Next, create a new 'MENU' resource. This one will contain the items that appear in the submenu. You create it and edit it as you would any other 'MENU'. Don't, however, add its ID to the 'MBAR' resource. Figure 7–13 shows an example submenu.

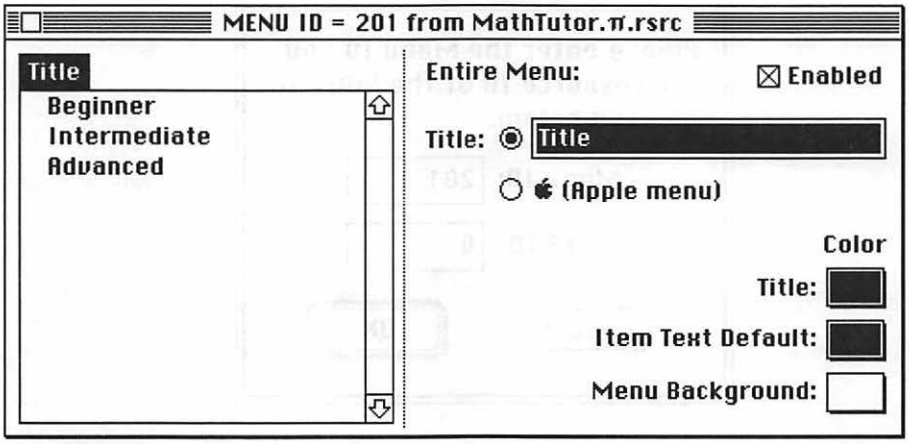


Figure 7–13. *A 'MENU' resource to be used as a submenu*

When you create the new 'MENU' by selecting "Create New Resource" from ResEdit's Resource menu, it might not give the new 'MENU' the same ID you specified in the previous 'MENU'—201 in Figure 7–12. Changing the 'MENU' ID is a two step process. First, select "Get Info" from ResEdit's File menu and change the ID there as shown in Figure 7–14. Then select "Edit Menu & MDEF ID" from the MENU menu and change the Menu ID there as well. Figure 7–15 shows this.

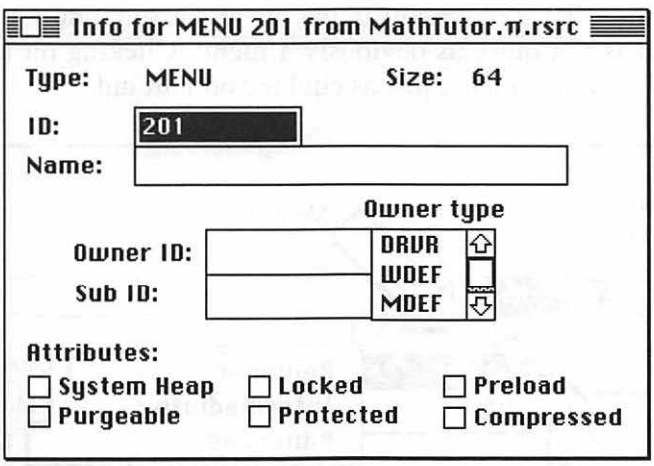


Figure 7–14. *Changing a 'MENU' ID in ResEdit's Get Info dialog*

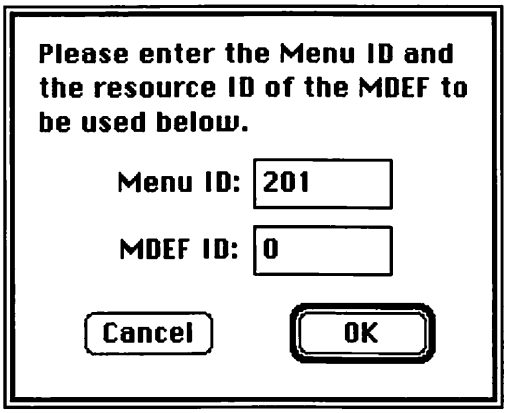


Figure 7–15. *Changing a 'MENU' ID in ResEdit's Edit Menu & MDEF ID Dialog*

That's it for resource changes. Now it's on to the source code.

Setting up the hierarchical menu

When you give a menu item a submenu you are, in effect, changing the item from a menu item to a menu. In Figure 7–16, the Edit menu is obviously a menu. Clicking on it displays the drop-down menu containing what appears to be three menu items: New, Lesson, and Quit. But the Lesson item is not quite as obviously a menu. Clicking on Lesson also displays a drop-down menu, just as clicking on Edit did.

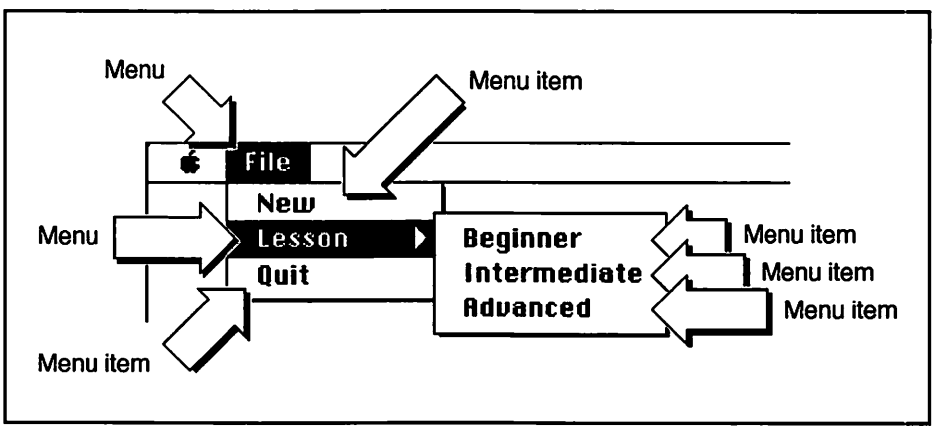


Figure 7–16. *Menus and menu items*

You know that when you set up your program's menu bar with *GetNewMBar()*, a menu list is created that contains a handle to each item. You can then obtain a handle to a menu using *GetMHandle()*:

```
menu_bar_handle = GetNewMBar( MENU_BAR_ID );

File_Menu  = GetMHandle( FILE_MENU_ID );
```

GetNewMBar() reads in the 'MENU' descriptions of the menus that will appear in the menu bar, the 'MENU's listed in the 'MBAR' resource. It also notes the menu ID of any submenus. It does not, however, read in the description of submenus. To read in that, you use *GetMenu()*. Then insert the submenu into the menu list using *InsertMenu()*. Here's an example:

```
MenuHandle  lesson_menu;

lesson_menu = GetMenu( LESSON_SUBMENU_ID );
InsertMenu( lesson_menu, -1 );
```

The -1 parameter used in *InsertMenu()* tells the Menu Manager that this menu is a submenu.

Now let's revise the *Set_Up_Menu_Bar()* routine introduced near the start of this chapter. This version adds the code for the insertion of a submenu into the File menu.

```
#define      MENU_BAR_ID         128
#define      APPLE_MENU_ID       128
#define      FILE_MENU_ID        129
#define      EDIT_MENU_ID        130
#define      LESSON_SUBMENU_ID   201

MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
MenuHandle   Edit_Menu;

void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;
   MenuHandle  submenu_handle;
```

```
    menu_bar_handle = GetNewMBar( MENU_BAR_ID );
    if ( menu_bar_handle == NIL )
       ExitToShell();

    SetMenuBar( menu_bar_handle );
    DisposHandle( menu_bar_handle );

    Apple_Menu = GetMHandle( APPLE_MENU_ID );
    File_Menu  = GetMHandle( FILE_MENU_ID );
    Edit_Menu  = GetMHandle( EDIT_MENU_ID );

    submenu_handle = GetMenu( LESSON_SUBMENU_ID );
    InsertMenu( submenu_handle, -1 );

    AddResMenu( Apple_Menu, 'DRVR' );

    DrawMenuBar();
}
```

You'll be pleased to find out that once a hierarchical menu is displayed, you handle it in the same way you handle traditional menus. Just include its 'MENU' ID in your *Handle_Menu_Choice()* routine, as done below.

```
#define      APPLE_MENU_ID      128
#define      FILE_MENU_ID       129
#define      EDIT_MENU_ID       130
#define      LESSON_SUBMENU_ID  201

void  Handle_Menu_Choice( long  menu_choice )
{
   [ extract menu and menu item from menu_choice ]

      switch ( the_menu )
      {
         case APPLE_MENU_ID:
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID:
            Handle_File_Choice( the_menu_item );
            break;

         case EDIT_MENU_ID:
```

```
            Handle_Edit_Choice( the_menu_item );
            break;

         case LESSON_SUBMENU_ID:
            Handle_Lesson_Choice( the_menu_item );
            break;

      }
      HiliteMenu(0);
   }
}
```

Changing Menu Characteristics

When working with various Macintosh programs, you've noticed that menu items might occasionally change during the running of a program. A menu item may have a checkmark placed to the left of it, or the text of a menu item might change. The most common change in a menu item or an entire menu is being enabled or disabled, so I'll discuss that first.

Disabling and enabling menus and menu items

During the running of a program not all menu options apply to all situations. When a menu item is not applicable you should disable, or dim, the item to prevent the user from choosing it. The most common example of the disabling of a menu item is the Paste command in the Edit menu. If the user hasn't cut or copied anything the Clipboard will be empty, and there will be nothing to paste. That's when a program will disable the Paste command.

You can disable a single item within a menu or an entire menu. In either case, the user can still click on the menu name in the menu bar to drop down the menu. If the entire menu is disabled, then the name in the menu bar will dim, along with the name of every item in the menu. Figure 7–17 shows that case on the left side of the picture. Disabling a single item in the menu does just that; every other item in the menu and the menu name in the menu bar appear normal. That's shown on the right side of Figure 7–17.

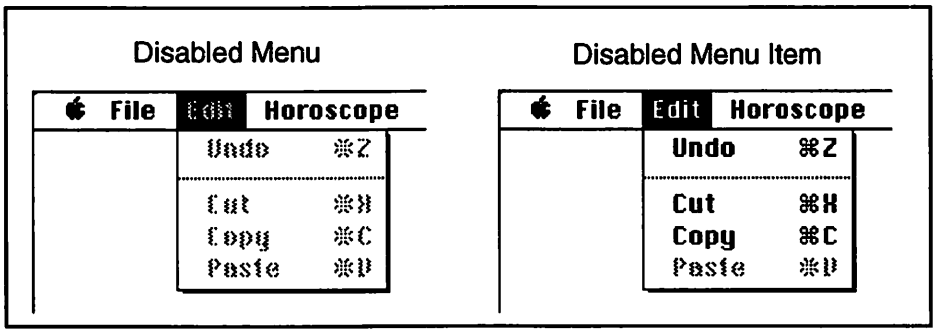


Figure 7-17. *A disabled menu and a disabled menu item*

To disable a single item use *DisableItem()*, passing a handle to the menu in which the item appears and the number of the item. Using the Edit menu of Figure 7–17, disable the Paste item. Remember, a dashed line in a menu counts as an item.

```
#define     EDIT_MENU_ID              130
#define     UNDO_ITEM                   1
/*          dashed line is 2nd item     2    */
#define     CUT_ITEM                    3
#define     COPY_ITEM                   4
#define     PASTE_ITEM                  5

MenuHandle  Edit_Menu;

Edit_Menu = GetMHandle( EDIT_MENU_ID);

DisableItem( Edit_Menu, PASTE_ITEM );

[ more code here ]

EnableItem( EditMenu, PASTE_ITEM );
```

As the above code shows, you enable an item using *EnableItem()*. Pass it the same parameters as *DisableItem()*.

Disabling an entire menu is just as easy as disabling a single menu item. In fact, you use the same Toolbox routine. The difference is in the value you pass as the second parameter. A menu item value of zero tells *DisableItem()* to disable the entire menu; that means the menu name in

the menu bar, as well as each item in the menu. Here's how you'd disable, then enable, the Edit menu.

```
#define    EDIT_MENU_ID              130
#define    UNDO_ITEM                   1
/*         dashed line is 2nd item     2    */
#define    CUT_ITEM                    3
#define    COPY_ITEM                   4
#define    PASTE_ITEM                  5

#define    ENTIRE_MENU                 0

MenuHandle  Edit_Menu;

Edit_Menu = GetMHandle( EDIT_MENU_ID);

DisableItem( Edit_Menu, ENTIRE_MENU );

[ more code here ]

EnableItem( EditMenu, ENTIRE_MENU );
```

Various circumstances can lead to the disabling and enabling of menu items. Every program may be different. Rather than scattering menu setting calls all about your source code, try the commonly used technique of grouping all the calls within one function.

For an example of menu highlighting, take a look at a hypothetical program named Horoscope; its menu bar is pictured back in Figure 7–17. Let's assume that under certain conditions, either the Enter Information item or the Show Forecast item may be disabled. When a condition occurs that requires a change in the state of a menu item—and that condition depends on the program I set a global *Boolean* variable appropriately. When there is a call to the menu-setting routine I check all these flags and set the state of each menu item accordingly.

```
#define    HOROSCOPE_MENU_ID         131
#define    ENTER_INFO_ITEM             1
/*         dashed line is 2nd item     2    */
#define    PRINT_SIGN_ITEM             3
#define    NO_PRINT_SIGN_ITEM          4
/*         dashed line is 5th item     5    */
```

```
#define    SHOW_FORECAST_ITEM          6

Boolean  Allow_Info_Input;
Boolean  Allow_Showing_Forecast;

void  Enable_Disable_Menu_Items( void )
{
   Horoscope_Menu = GetMHandle( HOROSCOPE_MENU_ID);

   if ( Allow_Info_Input == TRUE )
      EnableItem( Horoscope_Menu, ENTER_INFO_ITEM );
   else
      DisableItem( Horoscope_Menu, ENTER_INFO_ITEM );

   if ( Allow_Showing_Forecast == TRUE )
      EnableItem( Horoscope_Menu, SHOW_FORECAST_ITEM );
   else
      DisableItem( Horoscope_Menu, SHOW_FORECAST_ITEM );
}
```

The only time a user sees a menu item is when he clicks the mouse in the menu bar. So that's the only time you need to worry about each menu item being in its proper state. If you've set all flag variables at the appropriate places in the program, and if you place the call to *Enable_Disable_Menu_Items()* when the user clicks on the menu bar, then that's the one and only time you have to make the call.

As you saw earlier in this chapter, the Toolbox routine *MenuSelect()* is your means to handling all menu selections. If you call your menu-setting routine right before *MenuSelect()*, you'll be assured of having all your menu items in the proper state.

```
void  Handle_Mouse_Down( void )
{
   [ more code here ]

   switch ( the_part )
   {
      case inMenuBar:
         Enable_Disable_Menu_Items();
         menu_choice = MenuSelect( The_Event.where );
         Handle_Menu_Choice( menu_choice );
         break;
   }
}
```

Adding a checkmark to a menu item

A menu item can have a checkmark to the left of it to mark it as the current selection. Often a menu item that can be marked in this way is found in a group of two or more items. These items act as radio buttons in a dialog box—only one item in the grouping can be checked at any given time. Figure 7–18 shows a grouping of two menu items.

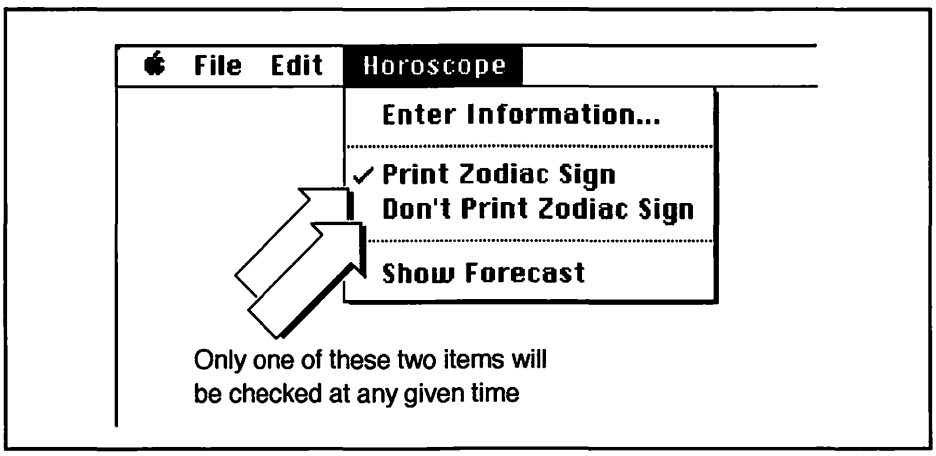


Figure 7–18. *A menu item with a checkmark*

Use the Toolbox routine *CheckItem()* to place a checkmark by an item or to remove a mark by an item. Pass *CheckItem()* a handle to the menu, the number of the item to check or uncheck, and a value of true to check the item or false to uncheck it. Here's an example that places a check by the third item in a menu.

```
#define     HOROSCOPE_MENU_ID          131
#define     ENTER_INFO_ITEM              1
/*          dashed line is 2nd item      2     */
#define     PRINT_SIGN_ITEM              3
#define     NO_PRINT_SIGN_ITEM           4
/*          dashed line is 5th item      5     */
#define     SHOW_FORECAST_ITEM           6

MenuHandle  Horoscope_Menu;

Horoscope_Menu = GetMHandle( HOROSCOPE_MENU_ID);

CheckItem( Horoscope_Menu, PRINT_SIGN_ITEM, TRUE );
```

The above shows the checking of an item. But you must also uncheck whichever item was checked previously. The example program in Figure 7–18 illustrates that process. A selection in any menu is handled by *Handle_Menu_Choice()*. From there a routine is called to handle the particular menu selected; in this case you end up at *Handle_Horoscope_Choice()*. Here's that routine:

```
void  Handle_Horoscope_Choice( int the_item )
{
   switch ( the_item )
   {
      case ENTER_INFO_ITEM:
         Open_Info_Dialog();
         break;

      case PRINT_SIGN_ITEM:
      case NO_PRINT_SIGN_ITEM:
         Handle_Menu_Checked_Item( the_item );
         break;

      case SHOW_FORECAST_ITEM:
         Open_Horoscope_Window();
         break;
   }
}
```

The two menu items involved in the checkmarking are both handled in the same way by *Handle_Menu_Checked_Item()*, shown next.

```
Boolean  Print_Sign_Flag;

void  Handle_Menu_Checked_Item( short item )
{
   if ( item == PRINT_SIGN_ITEM )
   {
      CheckItem( Horoscope_Menu, NO_PRINT_SIGN_ITEM, FALSE );
      Print_Sign_Flag = TRUE;
   }
```

```
    else
    {
       CheckItem( Horoscope_Menu, PRINT_SIGN_ITEM, FALSE );
       Print_Sign_Flag = FALSE;
    }

    CheckItem( Horoscope_Menu, item, TRUE );
}
```

I've passed *Handle_Menu_Checked_Item()* the number of the menu item selected; that is, the menu item to check. I first use that number to uncheck the item that was on. Then I check the passed-in item. What if the user selects a menu item that is already checked? The above code shows that *CheckItem()* will be called to uncheck an *already* unchecked item. Using *CheckItem()* to uncheck an unchecked item has no effect. The same applies to using it to check an already checked item. That's why this technique works. Figure 7–19 shows what happens if the Print Zodiac Sign item is already checked when it is again selected.

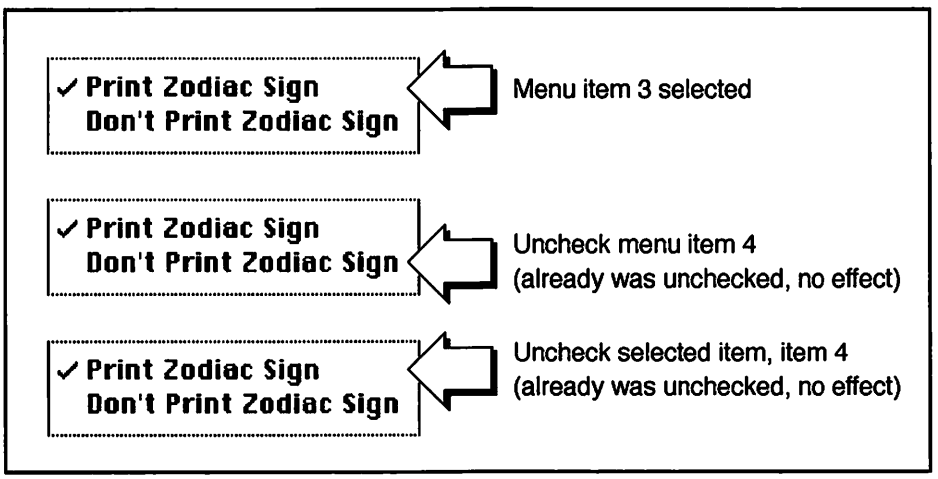


Figure 7–19. *Selecting an already checked item has no ill effect*

Figure 7–20 shows the case of Print Zodiac Sign in an unchecked state when this menu item is selected.

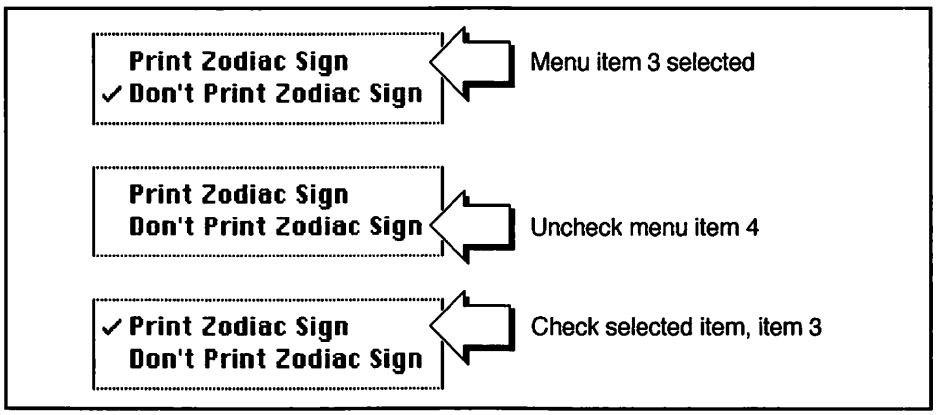


Figure 7-20. *Selecting an unchecked item checks it*

Notice that in *Handle_Menu_Checked_Item()* I set a global *Boolean* variable, *Print_Sign_Flag*, according to the selection made. This would be used at some other point in the program, perhaps to determine whether to print the user's astrological sign on each page of his horoscope when Show Forecast is selected.

One last point. If you're including menu items that get checked, make sure to check one and set any pertinent flags when you first set up the menu:

```
void  Set_Up_Menu_Bar( void )
{
   [ other menu code here ]

   Horoscope_Menu = GetMHandle( HOROSCOPE_MENU_ID );
   CheckItem( Horoscope_Menu, PRINT_SIGN_ITEM, TRUE );
   Print_Sign_Flag = TRUE;

   [ other menu code here ]
}
```

Changing the text of a menu item

You define the text that makes up each menu item in the 'MENU' resource of your program's resource file. If you want to change the text of a menu item during the execution of your program, use *SetItem()*. This Toolbox routine requires a handle to the affected menu, the item number

of the menu item to change, and a string that represents the new text. Here's an example that changes the text of a menu item from its resource definition of Enter Information... to Supply Missing Info...:

```
#define     HOROSCOPE_MENU_ID          131
#define     ENTER_INFO_ITEM              1

MenuHandle  Horoscope_Menu;

Horoscope_Menu = GetMHandle( HOROSCOPE_MENU_ID);
SetItem(Horoscope_Menu, ENTER_INFO_ITEM, "\pSupply Missing Info...");
```

Figure 7–21 shows the results. If you're concerned about the length of the new text exceeding the width of the dropped down menu—don't be. The Menu Manager knows to set the size of the menu according to the number of characters in the longest item string.

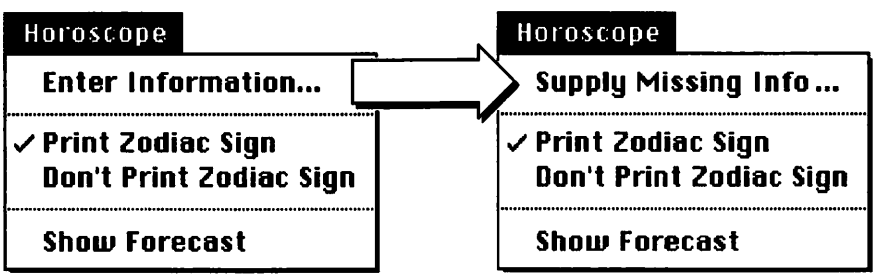


Figure 7–21. *Changing the text of a menu item*

If you like the liberal use of *#define* directives you might want to predefine the two titles the menu item might have:

```
#define     ENTER_INFO_STR       "\pEnter Information..."
#define     MISSING_INFO_STR     "\pSupply Missing Info..."

SetItem(Horoscope_Menu, ENTER_INFO_ITEM, MISSING_INFO_STR );
```

If you *really* want to do things right, you'll follow Apple's recommendation of not including text strings in your source code; they make it difficult to convert to another language. Instead, make each of the two menu item titles 'STR#' resources and store them in your program's resource file.

```
#define   STR_LIST_ID          128
#define   ENTER_INFO_STR         1
#define   MISSING_INFO_STR       2

Str255  the_str;

GetIndString( the_str, STR_LIST_ID, MISSING_INFO_STR );
SetItem(Horoscope_Menu, ENTER_INFO_ITEM, the_str );
```

Refer back to Chapter 3 for more information on this technique.

Of course, you won't be changing menu-item text randomly. In the example of 7–21, the decision to change the name of the menu item might be based on the amount of information the user entered in an Information Dialog. When the user closes the Information Dialog, the program can check for missing data and set the value of a global variable, *Data_Missing*, based on the results of this check.

```
[ Information Dialog closed here ]

[ check for missing user-supplied data ]

if ( Data_Missing == TRUE )
   SetItem(Horoscope_Menu, ENTER_INFO_ITEM, MISSING_INFO_STR );
else
   SetItem(Horoscope_Menu, ENTER_INFO_ITEM, ENTER_INFO_STR );
```

If you want to find out the current text of a menu item, use the sister routine of *SetItem()*: *GetItem()*. You were introduced to the routine earlier in this chapter in the discussion on opening desk accessories from the Apple menu.

Changing the style of a menu item

Now that you know you can change the text of a menu item, you may have guessed that you can also change the style of an item. The *SetItemStyle()* function is your means of doing this.

The Macintosh has a *Style* data type that contains the following constants: *plain, bold, italic, underline, outline, shadow, condense, and extend.* You can set a variable of type *Style* to any one of these values or,

to apply more than one style, add values. The following code sets a *Style* variable to bold and italic.

```
Style  item_style;

item_style = bold + italic;
```

With the style set, make a call to *SetItemStyle()*. Pass *SetItemStyle()* a *MenuHandle* and the item number corresponding to the item to change. A good time to do this is when you're setting up the menu bar. Here's an example that will display the fourth of four menu items in outline. Figure 7–22 shows the result.

```
#define    HOROSCOPE_MENU_ID    130
#define    ENTER_INFO_ITEM        1
#define    PRINT_SIGN_ITEM        2
#define    NO_PRINT_SIGN_ITEM     3
#define    SHOW_FORECAST_ITEM     4

MenuHandle  Horoscope_Menu;

Style  item_style;

Horoscope_Menu = GetMHandle( HOROSCOPE_MENU_ID);

item_style = outline;
SetItemStyle( Horoscope_Menu, SHOW_FORECAST_ITEM, item_style );
```

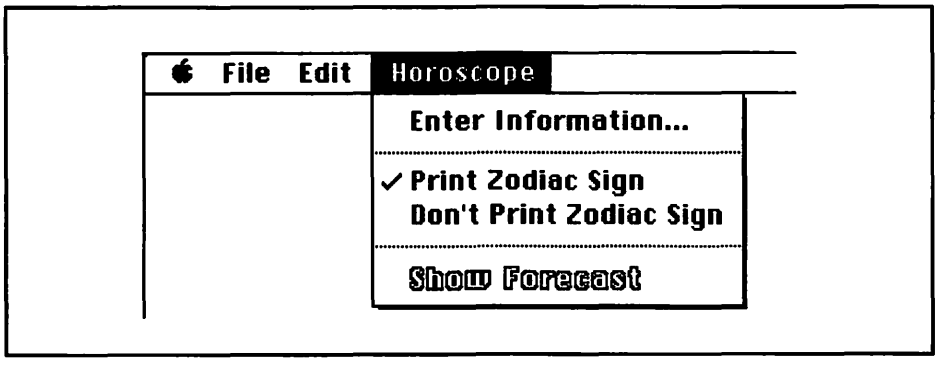


Figure 7–22. *A menu item with the outline style applied to it*

Perhaps your program allows the user to change a menu item's style. If so, you might not know just how a menu item is being displayed at any given time. In this case you can use *GetItemStyle()*. The parameters to this function are the same as those for *SetItemStyle()* except that the last one is a pointer to a *Style* rather than a *Style*. This allows the Toolbox to change its value, and it does. It will return a number that represents the menu item's current style or combination of styles. Here's a call to *GetItemStyle()*:

```
Style  item_style;

GetItemStyle( Horoscope_Menu, SHOW_CHART_ITEM, &item_style );
```

Each of the possible styles has a value, shown below. A menu item's current style is the sum of all the styles that have been applied to that item. As an example, if *GetItemStyle()* sets *item_style* to a value of 35, you know that the menu item is displayed as *condensed, italic, bold* (32 + 2 + 1).

```
plain       0
bold        1
italic      2
underline   4
outline     8
shadow     16
condense   32
extend     64
```

To determine which individual styles are in the sum, check for the largest value, *extend*. If it's there, subtract that value out and move on down the line. Here's an example that looks to see if a menu item has the *extend* or *condense* styles applied to it.

```
Boolean  extend_style   = FALSE;
Boolean  condense_style = FALSE;

if ( item_style >= extend )
{
   extend_style = TRUE;
   item_style -= extend;
}
if ( item_style >= condense )
```

```
{
   condense_style = TRUE;
   item_style -= condense;
}
[ add same tests for other styles ]
```

Editing Text in a Modal Dialog

Before System 7 a modal dialog owned the screen entirely. If the dialog appeared due to a menu selection, the menu name would invert in the menu bar, and all the menu names would dim. The user could not use the Edit menu to edit text in an edit text item. This situation is shown in the left of Figure 7–23.

System 7 adds a handy feature to the use of modal dialogs. If your application displays a modal dialog with one or more edit text items the system is now more generous. It will check to see if your program has a menu with the keyboard equivalents Command-X, Command-C, and Command-V. If your program does, the system will allow the user access to the Edit menu by enabling the menu along with the Cut, Copy, and Paste items. It will also take care of the editing, whether the user uses the menu or command key equivalents. The System 7 screen for this situation is shown on the right side of Figure 7–23

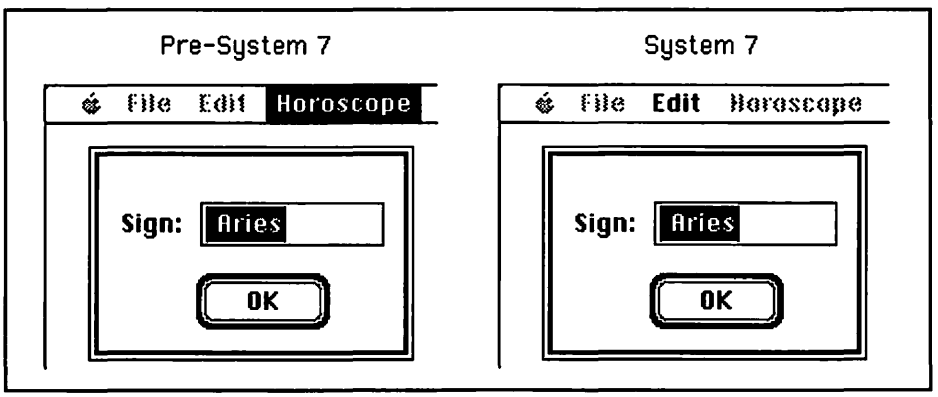


Figure 7–23. *The menu while a modal dialog is on the screen*

If you are absolutely sure that the program you're writing will never run on a pre-System 7 machine, you're all set. The system will take care of cut,

copy, and paste in a dialog. If your program might run on a Mac equipped with pre-System 7 software, but you aren't concerned with allowing the user access to cut and paste features in a dialog, you're again all set.

If you're a programmer who believes in accommodating the user—and as a Mac programmer, you must be—you'll want to make things easy on the user. That means giving the user the capability to edit text in a modal dialog, regardless of what system version the user has.

If your program is running under System 7, let the system do the work. Apple's written the code to handle the situation, so use it. Don't expend your time and energy trying to improve on what Apple engineers have already done! If your program is running on a Mac with pre-System 7 software, you can write a function to handle the dialog editing. Before discussing this function let's take a look at a simple way to see if the user has System 7.

Checking for System 7

How do you check to see what system the user has? Use the *Gestalt()* function. I introduced this powerful and handy Toolbox routine back in Chapter 4 when I wanted to see if the user had Color QuickDraw. I'll save you the effort of flipping pages by repeating that code here:

```
#include <GestaltEqu.h>

OSErr  err;
long   response;

err = Gestalt( gestaltQuickdrawVersion, &response );
```

Recall that *gestaltQuickdrawVersion* is defined, along with numerous other constants, in the header file *GestaltEqu.h*. You don't have to define it in your program.

Gestalt() can perform all sorts of inquiries into the system software and hardware of the Mac your program is running on; checking for Color QuickDraw is just one. Checking the system version is another. In the following code fragment I ask *Gestalt()* to return the system version in the variable response. If the response has a hexadecimal value of 0x0700,

or greater, your program is running on a Mac with a version of System 7. Set a global variable based on the findings of *Gestalt()*. Here's the code:

```
Boolean  System_7_Present;

OSErr  err;
long   response;
err = Gestalt( gestaltSystemVersion, &response );

if ( ( err == noErr ) && ( response >= 0x0700 ) )
   System_7_Present = TRUE;
else
   System_7_Present = FALSE;
```

Just a reminder—*Gestalt()* is covered in detail in Chapter 8. If you don't fully understand its workings now, don't be alarmed.

Once you know whether your program is running under System 7 you can make your decision whether to let the system handle dialog editing, or whether to handle it yourself with a filter function.

Modal dialog filter function

In the previous chapter you saw that the Toolbox routine *ModalDialog()* takes care of most of the work of handling a modal dialog. Here's a refresher:

```
[ open modal dialog ]

while ( all_done == FALSE )
{
   ModalDialog( NIL, &the_item );

   switch ( the_item )
   {
      case OK_BUTTON_ITEM:
         all_done = TRUE;
```

```
            break;

        [ other dialog items ]
    }
}
```

What I didn't tell you in the last chapter is that you can handle a modal dialog however you see fit—before *ModalDialog()* gets a crack at things. After you do the handling, you can then tell *ModalDialog()* to further handle things if you want.

The first parameter passed to *ModalDialog()* is the name of a *filter function* that does any necessary special handling of your dialog. This routine is optional. If you don't want to write one, pass in a nil pointer as you've done up to now. The time to use a filter function is when you have a dialog with special needs that *ModalDialog()* can't handle.

ModalDialog() handles update and activate events. It also intercepts user events and determines if an event occurred in an enabled item in the dialog box. If it did, it lets your program know which item was involved.

ModalDialog() will also track the user's actions in edit text boxes. It will flash the insertion bar in an edit text box, display typed characters, and invert selected text. What it won't do is handle text editing commands such as cut, copy, and paste. And that's a perfect application for a filter function.

To create a filter function, you write a function that performs the chores your dialog needs. The filter function always has three arguments: a pointer to the dialog box itself, a pointer to an *EventRecord*, and a pointer to a variable of type *short*. The return type of the function is always *Boolean*. And, as you learned in the Chapter 6 discussion on user items, the filter function needs to be prefaced with the *pascal* keyword. Here's a partial definition of a filter function called *Dialog_Edit_Filter()*:

```
pascal  Boolean  Dialog_Edit_Filter( DialogPtr dlog, EventRecord *event,
                                     short *item )
{
   [ handle edit commands here ]
}
```

Here's what a call to *ModalDialog()* would look like using a filter function:

```
ModalDialog( Dialog_Edit_Filter, &the_item );
```

Do you find it a little distressing that you can just use the name of a function as a parameter, without any parentheses or parameters? It's possible because *ModalDialog()* is expecting a pointer to a function as the first argument, not a call to a function. *ModalDialog()* uses the filter function name as a pointer to the function. It takes this pointer and uses it to go off into memory in search of your filter function.

When your program reaches a call to *ModalDialog()*, it branches off to the filter function. If the user performed some action that the filter function needs to handle, it does. It then returns a value of true to *ModalDialog()*. The question is, "Did the filter function handle the event?", and the answer is yes—or true. If the filter function handled the event, *ModalDialog()* doesn't have to. If it turns out that the user action did not require handling by the filter function, the function will return false. *ModalDialog()* knows it must then handle things itself. Figure 7–24 sums this all up. The shaded arrow is the point where the journey starts.

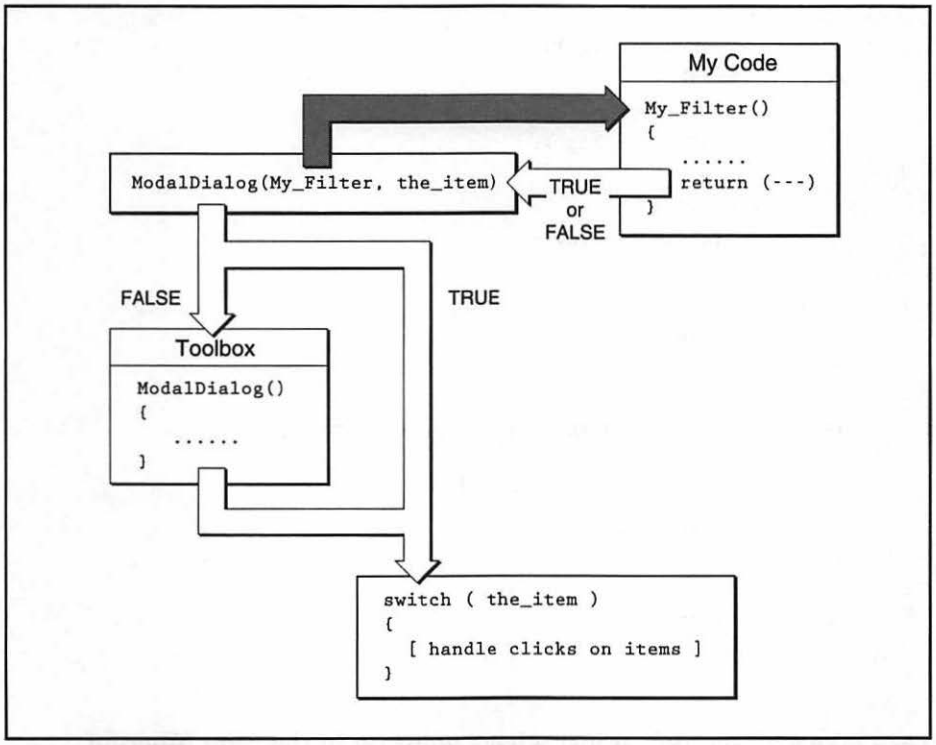


Figure 7–24. *Course of action when ModalDialog() uses a filter function*

Enough theory. Let's look at a real filter function. The *Dialog_Edit_Filter()* shown below allows the user to use the standard cut, copy, and paste keyboard aliases in the edit text items of a dialog. You can use it as is for any modal dialog you have.

```
pascal  Boolean  Dialog_Edit_Filter( DialogPtr dlog, EventRecord *event,
                                     short *item )
{
   char   chr;

   if ( event->what != keyDown )
      return ( FALSE );

   chr = event->message & charCodeMask;

   if ( ( event->modifiers & cmdKey ) != 0 )
   {
      switch ( chr )
      {
         case 'x':
            DlgCut ( dlog );
            break;
         case 'c':
            DlgCopy ( dlog );
            break;
         case 'v':
            DlgPaste ( dlog );
            break;
      }
      return ( TRUE );
   }

   if ( ( chr == RETURN_KEY ) || ( chr == ENTER_KEY ) )
   {
      *item = 1;
      return ( TRUE );
   }

   return ( FALSE );
}
```

Let's take a closer look at just what's going on in the filter function.

ModalDialog() passes the filter function each and every event it sees. The filter is only interested in events that involve the command key. If the event didn't involve a keystroke, let *ModalDialog()* handle it. So the very first thing the filter does is check to see if the event is a keystroke. If it isn't, the filter is through. It bails out and passes back a value of false; the event was not processed.

If the event survives the first test, it's an event involving a keystroke. The next step is to determine which key was pressed. This is done in the same way as *Handle_Keystroke()* did it. That routine was covered in the discussion on keyboard aliases:

```
chr = event->message & charCodeMask;
```

In *Handle_Keystroke()* the above line looked like this:

```
chr = The_Event.message & charCodeMask;
```

Remember your C? A structure member is accessed using the structure member operator, commonly called the dot:

```
EventRecord  The_Event;

The_Event.message
```

If you're working with a pointer to a structure, as is the case here, you must use the structure pointer operator: a minus sign followed by the greater than symbol:

```
EventRecord  *event;

event->message
```

Now, you want to see if the command key was pressed. This too was done back in *Handle_Keystroke()*. It involves looking at the *modifiers* member of the *EventRecord*.

```
if ( ( event->modifiers & cmdKey ) != 0 )
```

If the command key was pressed, the filter might actually be doing some real work! If the character key pressed along with the command key is either an

'x', a 'c', or a 'v', the user is attempting to edit the contents of a edit text item. A pointer to that item, by the way, was passed to the filter. To perform the actual editing in the dialog, the filter uses the Toolbox routines *DlgCut()*, *DlgCopy()*, and *DlgPaste()*. The event is handled, so the filter returns true.

The last test the filter makes is to see if the key that was pressed was either the return key or the enter key. If it was either, the filter treats the event as if there was a mouse-click on item number 1, the Done or OK button. It does this by setting the passed item variable to 1. That's why a pointer was passed in—so the filter could change its value.

If none of the above cases applied to the event, the event wasn't handled, so the filter returns a value of false to let *ModalDialog()* handle things.

If that breakdown of the filter function seemed a bit wordy, then you know it's time for a figure. Figure 7–25 sums it all up.

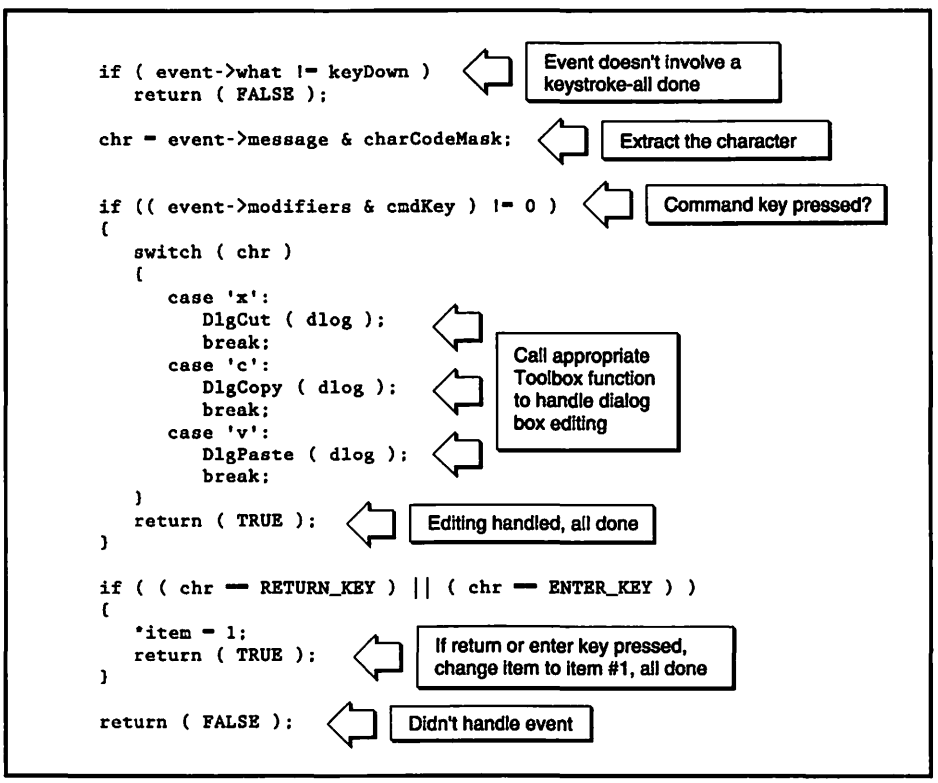
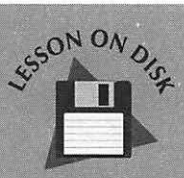


Figure 7–25. *Closer look at a dialog editing filter function*

Earlier I said that if the user has System 7 you should let the system handle editing in a dialog. I then showed how to use *Gestalt()* to test for the presence of System 7 and set a global *Boolean* variable using the test result. Now's the time to make use of this work. Use the *Boolean* variable in your dialog handling routine to determine whether to pass *ModalDialog()* the filter function or go it alone. Here's how.

```
[ open modal dialog ]

while ( all_done == FALSE )
{
   if ( System_7_Present == TRUE )
      ModalDialog( NIL, &the_item );
   else
      ModalDialog( Dialog_Edit_Filter, &the_item );

   switch ( the_item )
   {
      case OK_BUTTON_ITEM:
         all_done = TRUE;
         break;

      [ other dialog items ]
   }
}
```



Lesson 7-2: Filter Functions

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Chapter Program: *MenuMaster*

The example program for this chapter is *MenuMaster*. When you run the program you'll see a menu bar with three menus in it. Of most interest will be the File menu, shown in Figure 7–26.

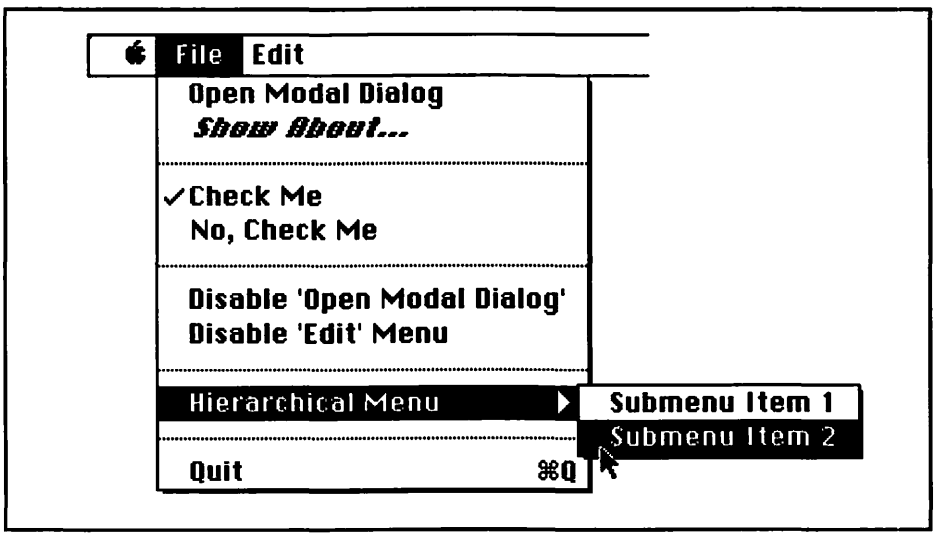


Figure 7–26. *MenuMaster's file menu*

A good deal of this chapter was devoted to demonstrating various techniques for changing the look of menu items. *MenuMaster* shows how these techniques work. It does all of the following:

- Includes the Apple menu in the menu bar.
- Enables and disables a menu item.
- Enables and disables an entire menu.
- Places a checkmark by menu items.
- Changes the text of menu items.
- Changes the style of a menu item.
- Displays a hierarchical menu.
- Uses a keyboard equivalent for a menu item.

Figure 7–27 shows the File menu of *MenuMaster* after a few of the menu items have been changed.

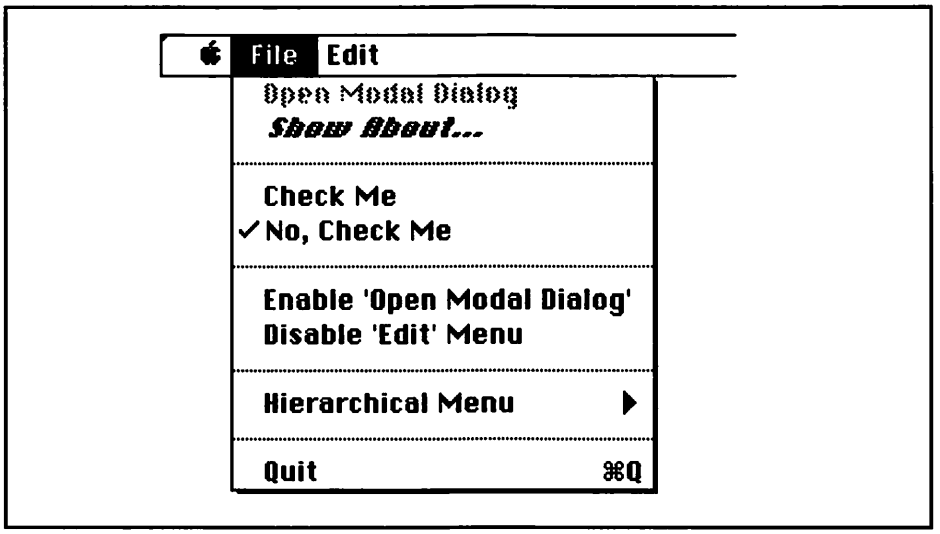


Figure 7-27. *MenuMaster's File menu, with a few items changed*

Selecting the "About MenuMaster" item from the Apple menu displays an alert that tells a little (very little) about the program. That alert is shown in Figure 7–28.

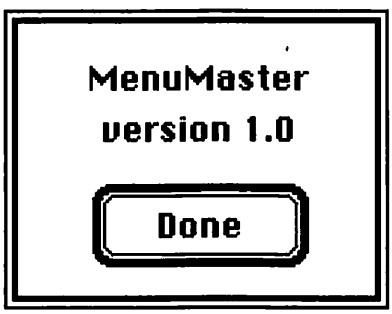


Figure 7-28. *The 'About MenuMaster' alert*

Selecting "Open Modal Dialog" from the File menu opens the modal dialog shown in Figure 7–29. Earlier I discussed allowing the user access to cut, copy, and paste commands while a modal dialog is on the screen. With this dialog box, *MenuMaster* demonstrates just how to do that.

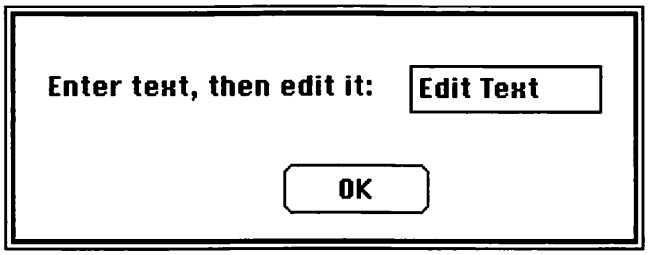


Figure 7–29. *MenuMaster's modal dialog, with editing capabilities*

The "Show About" item in the File menu displays the very same alert that the "About MenuMaster" item in the Apple menu displays.

When you choose the "Check Me" or "No, Check Me!", that item will receive a checkmark by it.

The "Disable 'Open Modal Dialog'" menu item does just that. Selecting it disables the first item in the File menu. With the first menu item now disabled, it would now be more appropriate if the text of the "Disable 'Open Modal Dialog'" read "Enable 'Open Modal Dialog'"—and it does.

The "Disable 'Edit' Menu" item works in the same manner as the previous item. It, however, disables an entire menu rather than just a single menu item.

The menu titled "Hierarchical Menu" is exactly that. It has two items in its submenu: Submenu Item 1 and Submenu Item 2. Each opens an alert that displays which choice was made. The alert for the first submenu item is shown in Figure 7–30.

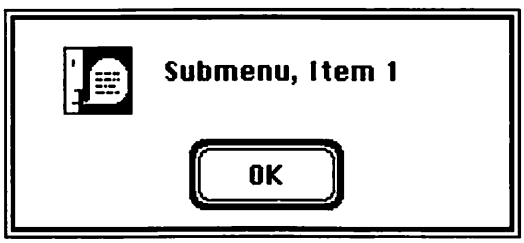


Figure 7–30. *A hierarchical submenu selection displays an alert*

The last menu item in the File menu is "Quit". You can use the keyboard equivalent Command-Q to quit the program.

Program resources: *MenuMaster.π.rsrc*

MenuMaster demonstrates menus in a Macintosh program and gives you a quick review of the previous chapter. The program uses two alerts and one dialog box. Figure 7–31 shows the five resource types used by the application.

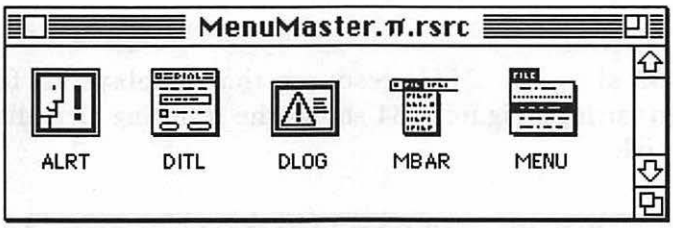


Figure 7–31. *The resource file for* MenuMaster

The two 'ALRT's have IDs of 128 and 129. So do their corresponding 'DITL' resources. They're shown in Figure 7–32.

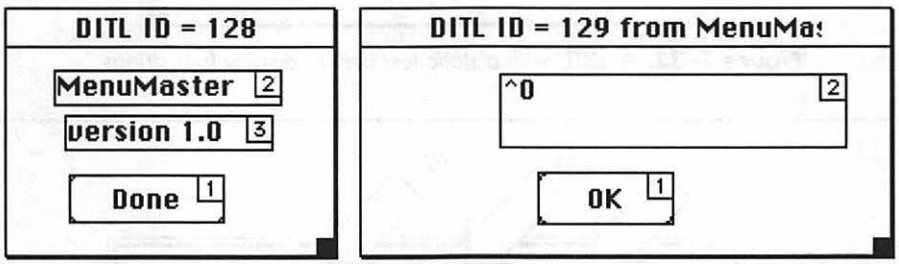


Figure 7–32. *The 'DITL' resources for* MenuMaster's *two 'ALRT's*

Of particular note is the strange "^0" text in item 2 in 'DITL' 129 in Figure 7–32. The alert that uses this 'DITL' is displayed by *MenuMaster* when either of the two items in its hierarchical submenu are selected. But rather than displaying "^0", the text in item 2 will be either "Submenu, Item 1" or "Submenu, Item 2." How do you use one alert to display different strings on different occasions? The answer is simple and clever and involves just one Toolbox call: *ParamText()*.

You pass *ParamText()* four strings. Your program will retain these four strings and use them in any alert or dialog that has one or more static items. How does it know which string to use in which item? The text of the static text item, defined when you create the 'DITL', must be one or more of the following: "^0", "^1", "^2", "^3". Your program will substitute the four *ParamText()* strings for each of these "^" strings. Here's an example:

```
#define    ALERT_ID      128

ParamText("\pMonday ", "\pTuesday", "\pWednesday ", "\pThursday");
Alert( ALERT_ID, NIL );
```

Figure 7–33 shows a 'DITL' resource that displays all four of the *ParamText()* strings. Figure 7–34 shows the resulting alert displayed for the above code.

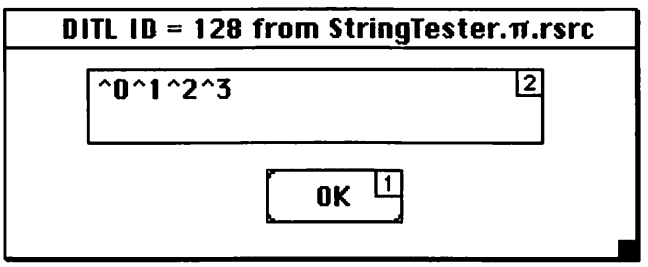


Figure 7–33. *A 'DITL' with a static text item to display four strings*

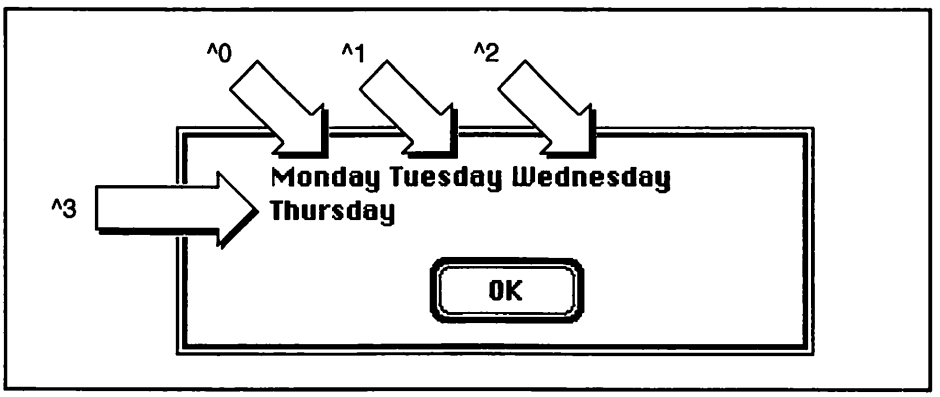


Figure 7–34. *The 'DITL' resources for MenuMaster's two 'ALRT's*

Now, what would happen if we called *ParamText()* again, substituting different strings, and then displayed the same alert? Here's the code:

```
#define    ALERT_ID      128

ParamText("\pMonday ", "\pTuesday", "\pWednesday ", "\pThursday");
Alert( ALERT_ID, NIL );
ParamText("\pFriday ", "\pSaturday ", "\pSunday ", "\p");
Alert( ALERT_ID, NIL );
```

First the alert in Figure 7–34 would be displayed. After clicking the OK button, the alert shown in Figure 7–35 would be displayed. Remember, both these alerts are using the same 'DITL'—the one pictured back in Figure 7–33.

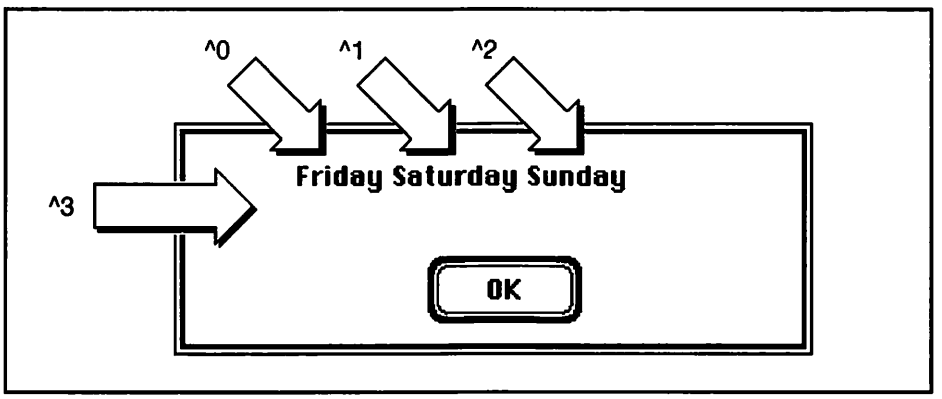


Figure 7–35. *The 'DITL' resources for MenuMaster's two 'ALRT's*

Notice in Figure 7–35 that only three strings seem to be displayed. The call to *ParamText()* defined the fourth string, the one to be displayed in the ^3 spot, as a null string—"\p".

If you look back a few pages you'll see that this whole discussion started with Figure 7–32. In that figure we showed you 'DITL' 129, which contained a static text item with the string "^0" in it. This 'DITL' will be used in an alert that will substitute a single string—for the "^0"—into the static text item.

To the third and final 'DITL' resource is used for the dialog box displayed when the user selects "Open Modal Dialog" from the File menu. It's pictured in Figure 7–36.

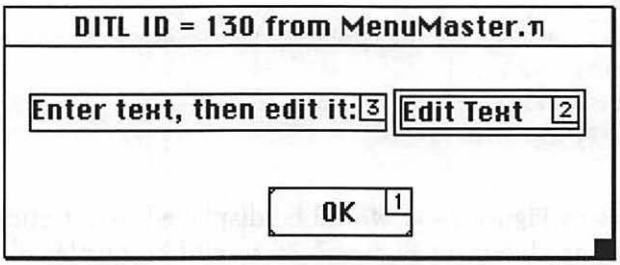


Figure 7–36. *The 'DITL' for* MenuMaster*'s modal dialog*

Now, to the menu-related resources, *MenuMaster*'s 'MBAR' resource is pictured in Figure 7–37. You can tell from the figure that the program has three menus—and, thus, three 'MENU' resources.

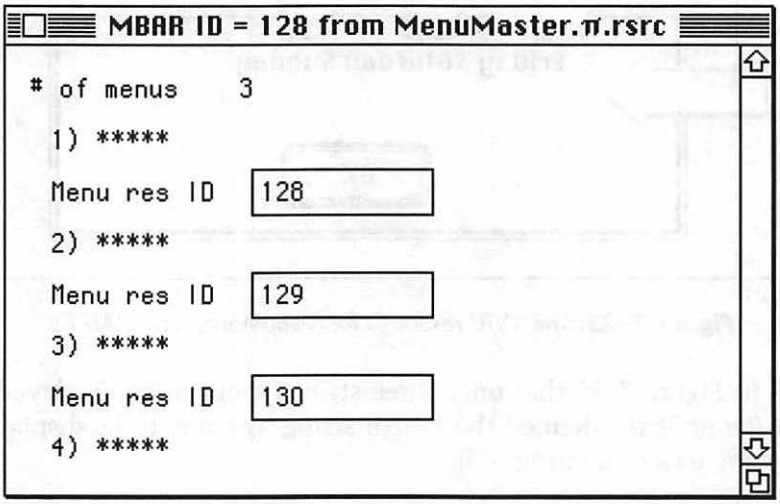


Figure 7–37. *The 'MBAR' for* MenuMaster

Figure 7–38 shows the three 'MENU' resources found in *MenuMaster*'s resource file.

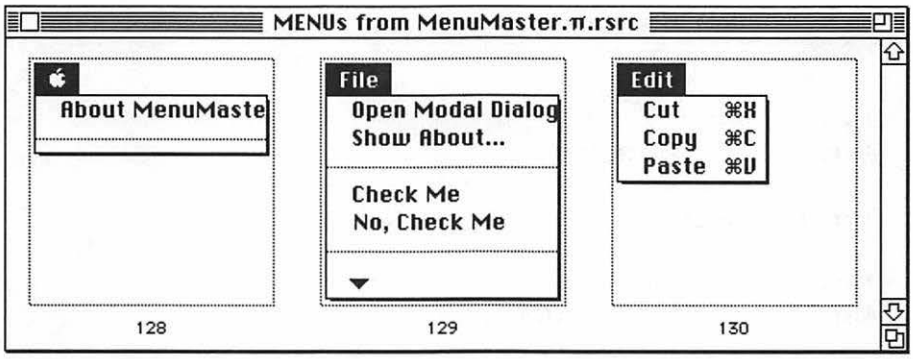


Figure 7-38. *The three 'MENU' resources for* MenuMaster

Program listing: *MenuMaster.c*

```
/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include <Traps.h>
#include <GestaltEqu.h>


/*++++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void    Initialize_Toolbox( void );
void    Initialize_Variables( void );
void    Set_Up_Menu_Bar( void );
void    Handle_One_Event( void );
void    Handle_Keystroke( void );
void    Handle_Mouse_Down( void );
void    Handle_Menu_Choice( long );
void    Handle_Apple_Choice( int );
void    Handle_File_Choice( int );
void    Open_Modal_Dialog( void );
pascal  Boolean Dialog_Edit_Filter( DialogPtr, EventRecord *, short * );
void    Handle_Hierarchical_Menu( int );
void    Handle_Menu_Checked_Item( short );
void    Handle_Disable_Edit_Item( void );
void    Handle_Disable_Open_Dialog_Item( void );


/*++++++++++++++++++   Define global constants   +++++++++++++++++*/
```

```
#define    ABOUT_ALERT_ID                128
#define    INFO_ALERT_ID                 129

#define    DIALOG_ID                     130
#define       OK_BUTTON_ITEM               1

#define    MENU_BAR_ID                   128

#define    APPLE_MENU_ID                 128
#define       SHOW_ABOUT_1_ITEM            1

#define    FILE_MENU_ID                  129
#define       OPEN_DIALOG_ITEM             1
#define       SHOW_ABOUT_2_ITEM            2
/*            ------------------           -  < dashed line       */
#define       CHECK_ME_ITEM                4
#define       NO_CHECK_ME_ITEM             5
/*            ------------------           -  < dashed line       */
#define       DISABLE_OPEN_DIALOG_ITEM     7
#define       DISABLE_EDIT_MENU_ITEM       8
/*            ------------------           -  < dashed line       */
/*            ++++++++++++++++++++         -  < hierarchical menu */
/*            ------------------           -  < dashed line       */
#define       QUIT_ITEM                   12

#define    SUBMENU_ID                    201
#define       SUBMENU_ITEM_1               1
#define       SUBMENU_ITEM_2               2

#define    EDIT_MENU_ID                  130
/*            ------------------           -  < item  1 is Cut    */
/*            ------------------           -  < item  2 is Copy   */
/*            ------------------           -  < item  3 is Paste  */

#define    ENTIRE_MENU                     0
#define    RETURN_KEY              (char)0x0D
#define    ENTER_KEY               (char)0x03

#define    NIL                            0L
#define    IN_FRONT           (WindowPtr)-1L
#define    REMOVE_EVENTS                   0
#define    SLEEP_TICKS                    0L
#define    MOUSE_REGION                   0L
```

```
/*++++++++++++++++   Define global variables   ++++++++++++++++++*/

Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
MenuHandle   Edit_Menu;
Boolean      Check_Me_Checked = FALSE;
Boolean      Open_Dialog_Disabled = FALSE;
Boolean      Edit_Menu_Disabled = FALSE;
Boolean      System_7_Present;


/*++++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void  main( void )
{
   Initialize_Toolbox();
   Initialize_Variables();

   Set_Up_Menu_Bar();

   while ( All_Done == FALSE )
      Handle_One_Event();
}


/*++++++++++++++++++  Initialize the Toolbox  ++++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*++++++++++++  Initialize some of our variables  ++++++++++++*/
```

```
void  Initialize_Variables( void )
{
   OSErr  err;
   long   response;

   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));

   err = Gestalt(gestaltSystemVersion, &response);

   if ((err == noErr) && (response >= 0x0700))
      System_7_Present = TRUE;
   else
      System_7_Present = FALSE;

   InitCursor();
}


/*+++++++++++++++  Set up menu bar and menus in it  +++++++++++++++*/

void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;
   MenuHandle  submenu_handle;
   Style       item_style;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      ExitToShell();

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   Apple_Menu = GetMHandle( APPLE_MENU_ID );
   File_Menu  = GetMHandle( FILE_MENU_ID );
   Edit_Menu  = GetMHandle( EDIT_MENU_ID );

   submenu_handle = GetMenu( SUBMENU_ID );
   InsertMenu( submenu_handle, -1 );

   item_style = bold + italic;
   SetItemStyle( File_Menu, SHOW_ABOUT_2_ITEM, item_style );
```

```
   CheckItem( File_Menu, NO_CHECK_ME_ITEM, TRUE );

   AddResMenu( Apple_Menu, 'DRVR' );

   DrawMenuBar();
}


/*++++++++++++++++++++  Handle a single event  ++++++++++++++++++*/

void  Handle_One_Event( void )
{
   Boolean  event_was_dialog;

   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
      case keyDown:
         Handle_Keystroke();
         break;

      case mouseDown:
         Handle_Mouse_Down();
         break;
      }
}


/*++++++++++++++++++++  Handle a keystroke  ++++++++++++++++++*/

void  Handle_Keystroke( void )
{
   short  chr;
   long   menu_choice;

   chr = The_Event.message & charCodeMask;

   if ( ( The_Event.modifiers & cmdKey ) != 0 )
```

```
    {
        if ( The_Event.what != autoKey )
        {
            menu_choice = MenuKey( chr );
            Handle_Menu_Choice( menu_choice );
        }
    }
}


/*++++++++++++  Handle a click of the mouse button  ++++++++++++++*/

void  Handle_Mouse_Down( void )
{
    WindowPtr  the_window;
    short      the_part;
    long       menu_choice;

    the_part = FindWindow( The_Event.where, &the_window );

    switch ( the_part )
    {
        case inMenuBar:
            menu_choice = MenuSelect( The_Event.where );
            Handle_Menu_Choice( menu_choice );
            break;
    }
}


/*++++++++++++++  Respond to a click in a menu  ++++++++++++++++++*/

void  Handle_Menu_Choice( long  menu_choice )
{
    int  the_menu;
    int  the_menu_item;

    if ( menu_choice != 0 )
    {
        the_menu = HiWord( menu_choice );
        the_menu_item = LoWord( menu_choice );

        switch ( the_menu )
        {
            case APPLE_MENU_ID:
```

```
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID:
            Handle_File_Choice( the_menu_item );
            break;

         case SUBMENU_ID:
            Handle_Hierarchical_Menu( the_menu_item );
            break;

         case EDIT_MENU_ID:
            break;
      }
      HiliteMenu(0);
   }
}


/*++++++++++++  Handle selection from 'Apple' menu  ++++++++++++++*/

void  Handle_Apple_Choice( int the_item )
{
   Str255  desk_acc_name;
   int     desk_acc_number;

   switch ( the_item )
   {
      case SHOW_ABOUT_1_ITEM :
         Alert( ABOUT_ALERT_ID, NIL );
         break;

      default :
         GetItem( Apple_Menu, the_item, desk_acc_name );
         desk_acc_number = OpenDeskAcc( desk_acc_name );
         break;
   }
}


/*++++++++++++  Handle selection from 'File' menu  +++++++++++++++*/

void  Handle_File_Choice( int the_item )
{
   switch ( the_item )
```

```
    {
        case OPEN_DIALOG_ITEM:
            Open_Modal_Dialog();
            break;

        case SHOW_ABOUT_2_ITEM:
            Alert( ABOUT_ALERT_ID, NIL );
            break;

        case CHECK_ME_ITEM:
        case NO_CHECK_ME_ITEM:
            Handle_Menu_Checked_Item( the_item );
            break;

        case DISABLE_OPEN_DIALOG_ITEM:
            Handle_Disable_Open_Dialog_Item();
            break;

        case DISABLE_EDIT_MENU_ITEM:
            Handle_Disable_Edit_Item();
            break;

        case QUIT_ITEM:
            All_Done = TRUE;
            break;
    }
}


/*+++++++  Open a modal dialog to test 'Edit' menu items  ++++++*/

void  Open_Modal_Dialog( void )
{
    DialogPtr  the_dialog;
    short      the_item;
    Boolean    all_done = FALSE;

    the_dialog = GetNewDialog( DIALOG_ID, NIL, IN_FRONT );
    ShowWindow( the_dialog );

    while ( all_done == FALSE )
    {
      if ( System_7_Present == TRUE )
          ModalDialog( NIL, &the_item );
       else
```

```
        ModalDialog( Dialog_Edit_Filter, &the_item );

      switch ( the_item )
      {
         case OK_BUTTON_ITEM:
            all_done = TRUE;
            break;
      }
   }
   DisposDialog( the_dialog );
}


/*+++  Filter function for ModalDialog of Open_Modal_Dialog  +++*/

pascal  Boolean Dialog_Edit_Filter( DialogPtr dlog, EventRecord *event, short *item )
{
   char   chr;

   if ( event->what != keyDown )
      return ( FALSE );

   chr = event->message & charCodeMask;

   if ( ( event->modifiers & cmdKey ) != 0 )
   {
      switch ( chr )
      {
         case 'x':
            DlgCut ( dlog );
            break;
         case 'c':
            DlgCopy ( dlog );
            break;
         case 'v':
            DlgPaste ( dlog );
            break;
      }
      return ( TRUE );
   }

   if ( ( chr == RETURN_KEY ) || ( chr == ENTER_KEY ) )
   {
      *item = 1;
      return ( TRUE );
```

```
    }

    return ( FALSE );
}


/*++++++++++++  Handle Check Me item from File menu  ++++++++++++*/

void  Handle_Menu_Checked_Item( short item )
{
    if ( item == CHECK_ME_ITEM )
    {
        CheckItem( File_Menu, NO_CHECK_ME_ITEM, FALSE );
        Check_Me_Checked = TRUE;
    }
    else
    {
        CheckItem( File_Menu, CHECK_ME_ITEM, FALSE );
        Check_Me_Checked = FALSE;
    }

    CheckItem( File_Menu, item, TRUE );
}


/*+++  Handle Disable Open Modal Dialog item from File menu  +++*/

void  Handle_Disable_Open_Dialog_Item( void )
{
    if ( Open_Dialog_Disabled == TRUE )
    {
        EnableItem( File_Menu, OPEN_DIALOG_ITEM );
        SetItem( File_Menu, DISABLE_OPEN_DIALOG_ITEM,
                "\pDisable 'Open Modal Dialog'");
        Open_Dialog_Disabled = FALSE;
    }
    else
    {
        DisableItem( File_Menu, OPEN_DIALOG_ITEM );
        SetItem( File_Menu, DISABLE_OPEN_DIALOG_ITEM,
                "\pEnable 'Open Modal Dialog'");
        Open_Dialog_Disabled = TRUE;
    }
}
```

```
/*+++++++  Handle Disable Edit Menu item from File menu  +++++++*/

void  Handle_Disable_Edit_Item( void )
{
   if ( Edit_Menu_Disabled == TRUE )
   {
      EnableItem( Edit_Menu, ENTIRE_MENU );
      DrawMenuBar();
      SetItem( File_Menu, DISABLE_EDIT_MENU_ITEM, "\pDisable 'Edit' Menu");
      Edit_Menu_Disabled = FALSE;
   }
   else
   {
      DisableItem( Edit_Menu, ENTIRE_MENU );
      DrawMenuBar();
      SetItem( File_Menu, DISABLE_EDIT_MENU_ITEM, "\pEnable 'Edit' Menu");
      Edit_Menu_Disabled = TRUE;
   }
}


/*++++++  Handle selection from 'Hierarchical Menu' menu  ++++++*/

void  Handle_Hierarchical_Menu( int the_item )
{
   switch ( the_item )
   {
      case SUBMENU_ITEM_1:
         ParamText("\pSubmenu, Item 1", "\p", "\p", "\p");
         NoteAlert( INFO_ALERT_ID, NIL );
         break;

      case SUBMENU_ITEM_2:
         ParamText("\pSubmenu, Item 2", "\p", "\p", "\p");
         NoteAlert( INFO_ALERT_ID, NIL );
         break;
   }
}
```

Stepping through the code

Once again, it's time to step through the source code to see just what's going on.

The #define directives

All but three of the *#defines* new to this program are resource IDs or resource item numbers. If you want to make any changes to *MenuMaster*'s resource file, you only have to go to one place in the source code to make changes or additions—the *#defines* section.

The two 'ALRT' resources have IDs of *ABOUT_ALERT_ID* and *INFO_ALERT_ID*. The modal dialog has a 'DLOG' ID of *DIALOG_ID*. The OK button in that dialog has an item number of *OK_BUTTON_ITEM*.

The 'MBAR' has an ID of *MENU_BAR_ID*. There are three 'MENU's in it.

The first 'MENU' has an ID of *APPLE_MENU_ID*. The Apple menu has one item, *SHOW_ABOUT_1_ITEM*.

The second 'MENU', with an ID of *FILE_MENU_ID*, has 12 items, but four of them are dashed lines and one is a hierarchical menu. Those five don't get #defines. That leaves *OPEN_DIALOG_ITEM*, *SHOW_ABOUT_2_ITEM*, *CHECK_ME_ITEM*, *NO_CHECK_ME_ITEM*, *DISABLE_OPEN_DIALOG_ITEM*, *DISABLE_EDIT_MENU_ITEM*, and *QUIT_ITEM*.

The hierarchical menu that appears in the File menu has its own 'MENU' resource to define the items in it. The ID of that resource is *SUBMENU_ID*. The two items in it are *SUBMENU_ITEM_1* and *SUBMENU_ITEM_2*.

The final 'MENU' in the menu bar is the Edit menu, with a resource ID of *EDIT_MENU*. It has three items in it. As the source code will demonstrate, you won't be using any of them directly, so they don't require #defines.

Normally a call to *DisableItem()* disables a single menu item. If you pass the routine *ENTIRE_MENU* as a parameter, though, an entire menu will be disabled.

RETURN_KEY and *ENTER_KEY* are the character constants for the return key and the enter key. The filter function for *ModalDialog()* will use these.

```
#define     ABOUT_ALERT_ID              128
```

```
#define    INFO_ALERT_ID                 129

#define    DIALOG_ID                     130
#define       OK_BUTTON_ITEM               1

#define    MENU_BAR_ID                   128

#define    APPLE_MENU_ID                 128
#define       SHOW_ABOUT_1_ITEM            1

#define    FILE_MENU_ID                  129
#define       OPEN_DIALOG_ITEM             1
#define       SHOW_ABOUT_2_ITEM            2
/*            ------------------           -  < dashed line       */
#define       CHECK_ME_ITEM                4
#define       NO_CHECK_ME_ITEM             5
/*            ------------------           -  < dashed line       */
#define       DISABLE_OPEN_DIALOG_ITEM     7
#define       DISABLE_EDIT_MENU_ITEM       8
/*            ------------------           -  < dashed line       */
/*            ++++++++++++++++++++         -  < hierarchical menu */
/*            ------------------           -  < dashed line       */
#define       QUIT_ITEM                   12

#define    SUBMENU_ID                    201
#define       SUBMENU_ITEM_1               1
#define       SUBMENU_ITEM_2               2

#define    EDIT_MENU_ID                  130
/*            ------------------           -  < item  1 is Cut    */
/*            ------------------           -  < item  2 is Copy   */
/*            ------------------           -  < item  3 is Paste  */

#define    ENTIRE_MENU                     0
#define    RETURN_KEY              (char)0x0D
#define    ENTER_KEY               (char)0x03

#define    NIL                            0L
#define    IN_FRONT           (WindowPtr)-1L
#define    REMOVE_EVENTS                   0
#define    SLEEP_TICKS                    0L
#define    MOUSE_REGION                   0L
```

The global variables

Like the previous examples, *MenuMaster* uses *All_Done*, *Multifinder_Present*, and *The_Event* in dealing with events. The program puts up three menus. So that you can work with the menus, you'll want a global *MenuHandle* variable for each. They are *Apple_Menu*, *File_Menu*, *Edit_Menu*. *MenuMaster* will be toggling the text of some menu items. Simplistically, you'll be checking for something like this: if a menu says "A", make it "B". If it says "B", make it "A". These *Boolean* variables will keep track of the current state of three of the menu items: *Menu_Checked*, *Open_Dialog_Disabled*, and *Edit_Menu_Disabled*. Finally, *MenuMaster* checks to see if it's running on a Macintosh that has System 7. It uses *System_7_Present* to hold the result of this check.

```
Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
MenuHandle   Edit_Menu;
Boolean      Check_Me_Checked = FALSE;
Boolean      Open_Dialog_Disabled = FALSE;
Boolean      Edit_Menu_Disabled = FALSE;
Boolean      System_7_Present;
```

The main() function

I said early in this chapter that your program should put up the menu bar soon after starting. *MenuMaster* does just that. Right after the traditional initialization of the Toolbox and a few program variables, the menu goes up with a call to *Set_Up_Menu_Bar()*.

```
void  main( void )
{
   Initialize_Toolbox();
   Initialize_Variables();

   Set_Up_Menu_Bar();

   while ( All_Done == FALSE )
      Handle_One_Event();
}
```

Initializing variables

After initializing the Toolbox, *MenuMaster* calls *Initialize_Variables()* to give a few program globals their values. Here you use a call to *Gestalt()* to determine if the program is running under System 7. You'll use this information to decide if you'll let the system handle dialog box editing the modal dialog comes up. Here's the *Initialize_Variables()* routine:

```
void  Initialize_Variables( void )
{
   OSErr  err;
   long   response;

   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));

   err = Gestalt(gestaltSystemVersion, &response);

   if ((err == noErr) && (response >= 0x0700))
      System_7_Present = TRUE;
   else
      System_7_Present = FALSE;

   InitCursor();
}
```

Setting up the menu bar

MenuMaster calls *Set_Up_Menu_Bar()* to put the menu bar on the screen. This routine is so similar to the one by the same name developed in this chapter's Setting Up the Hierarchical Menu section, you'd swear I did a copy and paste. Me? Never! Seriously, though, just a few lines are new. I've added a *Style* variable and these three lines:

```
item_style = bold + italic;
SetItemStyle( File_Menu, SHOW_ABOUT_2_ITEM, item_style );

CheckItem( File_Menu, NO_CHECK_ME_ITEM, TRUE );
```

The second item in the File menu, the "Show About" item, appears in bold and italic. I add the styles I want, then call *SetItemStyle()* to make the style change. This is the only place I have to make the change; the

menu item will appear in this style for the remainder of the program's execution.

MenuMaster has two items that can receive a checkmark. The program starts with one of the items checked, so I do that here with a call to *CheckItem()*.

```
void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;
   MenuHandle  submenu_handle;
   Style       item_style;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      ExitToShell();

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   Apple_Menu = GetMHandle( APPLE_MENU_ID );
   File_Menu  = GetMHandle( FILE_MENU_ID );
   Edit_Menu  = GetMHandle( EDIT_MENU_ID );

   submenu_handle = GetMenu( SUBMENU_ID );
   InsertMenu( submenu_handle, -1 );

   item_style = bold + italic;
   SetItemStyle( File_Menu, SHOW_ABOUT_2_ITEM, item_style );

   CheckItem( File_Menu, NO_CHECK_ME_ITEM, TRUE );

   AddResMenu( Apple_Menu, 'DRVR' );

   DrawMenuBar();
}
```

Handling a keystroke

The *Handle_One_Event()* routine should be old-hat by now. This program's version has just one addition—the handling of a keystroke. I've included a case section for a *keyDown* event. There I call *Handle_Keystroke()*. This

routine appears exactly as it was developed in this chapter's Handling A Keystroke section.

```
void  Handle_One_Event( void )
{
   Boolean  event_was_dialog;

   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
      case keyDown:
         Handle_Keystroke();
         break;

      case mouseDown:
         Handle_Mouse_Down();
         break;
      }
}


void  Handle_Keystroke( void )
{
   short  chr;
   long   menu_choice;

   chr = The_Event.message & charCodeMask;

   if ( ( The_Event.modifiers & cmdKey ) != 0 )
   {
      if ( The_Event.what != autoKey )
      {
         menu_choice = MenuKey( chr );
         Handle_Menu_Choice( menu_choice );
      }
   }
}
```

Handling a click in the menu bar

A mouse click results in a call to *Handle_Mouse_Down()*, which in turn calls *Handle_Menu_Choice()*. Here's *Handle_Mouse_Down()*.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;
   long       menu_choice;

   the_part = FindWindow( The_Event.where, &the_window );

   switch ( the_part )
   {
      case inMenuBar:
         menu_choice = MenuSelect( The_Event.where );
         Handle_Menu_Choice( menu_choice );
         break;
   }
}
```

Regardless of the program, *Handle_Menu_Choice()* has the same form: use *HiWord()* and *LoWord()* to determine the selected menu and menu item, respectively. Then enter a *switch* statement that determines which menu-handling routine to branch to.

MenuMaster has three menus in the menu bar, yet there are four cases in the *switch*. That's because *MenuMaster* has a hierarchical menu; don't forget to include all hierarchical menus in the *switch*. Even though the user goes through the File menu to reach the hierarchical menu, it still acts as if it were a menu perched in the menu bar.

Notice that a click in the edit menu doesn't get any attention. That's because *MenuMaster* only uses the Edit menu when the modal dialog is open. When that's the case, either the system will handle things (if System 7 is present), or the filter function called by *ModalDialog()* will (if any other earlier system is running).

```
void  Handle_Menu_Choice( long  menu_choice )
{
   int  the_menu;
   int  the_menu_item;

   if ( menu_choice != 0 )
   {
      the_menu = HiWord( menu_choice );
      the_menu_item = LoWord( menu_choice );

      switch ( the_menu )
      {
         case APPLE_MENU_ID:
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID:
            Handle_File_Choice( the_menu_item );
            break;

         case SUBMENU_ID:
            Handle_Hierarchical_Menu( the_menu_item );
            break;

         case EDIT_MENU_ID:
            break;
      }
      HiliteMenu(0);
   }
}
```

A menu selection in the Apple menu brings you to *Handle_Apple_Choice()*. This is a typical "cut and paste" routine; it will appear, as is, in almost any program you write. What would make you change this routine? If you have more than one item in the menu, other than the desk accessories. Figure 7–39 gives an example.

```
void  Handle_Apple_Choice( int the_item )
{
   Str255  desk_acc_name;
   int     desk_acc_number;

   switch ( the_item )
   {
```

```
        case SHOW_ABOUT_1_ITEM :
           Alert( ABOUT_ALERT_ID, NIL );
           break;

        default :
           GetItem( Apple_Menu, the_item, desk_acc_name );
           desk_acc_number = OpenDeskAcc( desk_acc_name );
           break;
     }
  }
```

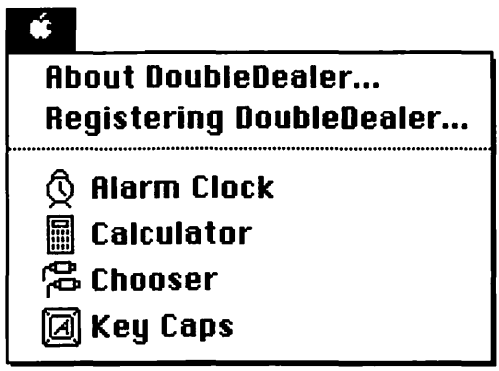


Figure 7–39. *Example of a "nonstandard" Apple menu*

A selection in the File menu sends you to *Handle_File_Choice()*. Typical of menu-handling routines, it isn't much more than a branching-off point. A "Show About" selection simply puts up the same alert that you used for the Apple menu's "About MenuMaster" item. Choosing "Quit" just sets the global variable *All_Done* to true. The other menu items are a bit more complicated, so they have their own routines.

```
void  Handle_File_Choice( int the_item )
{
   switch ( the_item )
   {
      case OPEN_DIALOG_ITEM:
         Open_Modal_Dialog();
         break;

      case SHOW_ABOUT_2_ITEM:
         Alert( ABOUT_ALERT_ID, NIL );
         break;
```

```
        case CHECK_ME_ITEM:
           Handle_Menu_Checked_Item();
           break;

        case DISABLE_OPEN_DIALOG_ITEM:
           Handle_Disable_Open_Dialog_Item();
           break;

        case DISABLE_EDIT_MENU_ITEM:
           Handle_Disable_Edit_Item();
           break;

        case QUIT_ITEM:
           All_Done = TRUE;
           break;
     }
}
```

Editing text in a modal dialog

A menu choice of Open Modal Dialog takes the program to a routine called *Open_Modal_Dialog()*. How's that for descriptive naming? The source code for this routine is pretty much straight out of the Modal Dialog Source Code section of Chapter 6.

There is one addition—the check for System 7 discussed earlier. If System 7 isn't present, you must supply the means for the user to use the keyboard equivalents for cut, copy, and paste. *Dialog_Edit_Filter()* does that. There's no need to discuss the filter function here—it appears just as it does several pages back.

```
void  Open_Modal_Dialog( void )
{
   DialogPtr  the_dialog;
   short      the_item;
   Boolean    all_done = FALSE;

   the_dialog = GetNewDialog( DIALOG_ID, NIL, IN_FRONT );
   ShowWindow( the_dialog );

   while ( all_done == FALSE )
   {
     if ( System_7_Present == TRUE )
```

```
            ModalDialog( NIL, &the_item );
         else
            ModalDialog( Dialog_Edit_Filter, &the_item );

         switch ( the_item )
         {
            case OK_BUTTON_ITEM:
               all_done = TRUE;
               break;
         }
      }
      DisposDialog( the_dialog );
}


pascal  Boolean Dialog_Edit_Filter( DialogPtr dlog, EventRecord *event,
                                    short *item )
{
   char   chr;

   if ( event->what != keyDown )
      return ( FALSE );

   chr = event->message & charCodeMask;

   if ( ( event->modifiers & cmdKey ) != 0 )
   {
      switch ( chr )
      {
         case 'x':
            DlgCut ( dlog );
            break;
         case 'c':
            DlgCopy ( dlog );
            break;
         case 'v':
            DlgPaste ( dlog );
            break;
      }
      return ( TRUE );
   }

   if ( ( chr == RETURN_KEY ) || ( chr == ENTER_KEY ) )
   {
      *item = 1;
```

```
        return ( TRUE );
    }

    return ( FALSE );
}
```

Checking a menu item

This chapter demonstrated how to use *CheckItem()* to either set or clear a checkmark by a menu item. *MenuMaster* uses this same technique.

```
void  Handle_Menu_Checked_Item( short item )
{
    if ( item == CHECK_ME_ITEM )
    {
        CheckItem( File_Menu, NO_CHECK_ME_ITEM, FALSE );
        Check_Me_Checked = TRUE;
    }
    else
    {
        CheckItem( File_Menu, CHECK_ME_ITEM, FALSE );
        Check_Me_Checked = FALSE;
    }

    CheckItem( File_Menu, item, TRUE );
}
```

Disabling and enabling a menu and menu item

If the user selects the File menu item "Disable 'Open Modal Dialog'" you check the global flag *Open_Dialog_Disabled* to see which state this item is already in. Whatever the state, you toggle it to its opposite state. This routine performs two tasks. It enables or disables the first item in the File menu, and then makes a call to *SetItem()* to change the text of the selected item to whatever title is appropriate. Figure 7–40 shows the two possible scenarios.

```
void  Handle_Disable_Open_Dialog_Item( void )
{
   if ( Open_Dialog_Disabled == TRUE )
   {
      EnableItem( File_Menu, OPEN_DIALOG_ITEM );
      SetItem( File_Menu, DISABLE_OPEN_DIALOG_ITEM,
               "\pDisable 'Open Modal Dialog'");
      Open_Dialog_Disabled = FALSE;
   }
   else
   {
      DisableItem( File_Menu, OPEN_DIALOG_ITEM );
      SetItem( File_Menu, DISABLE_OPEN_DIALOG_ITEM,
               "\pEnable 'Open Modal Dialog'");
      Open_Dialog_Disabled = TRUE;
   }
}
```

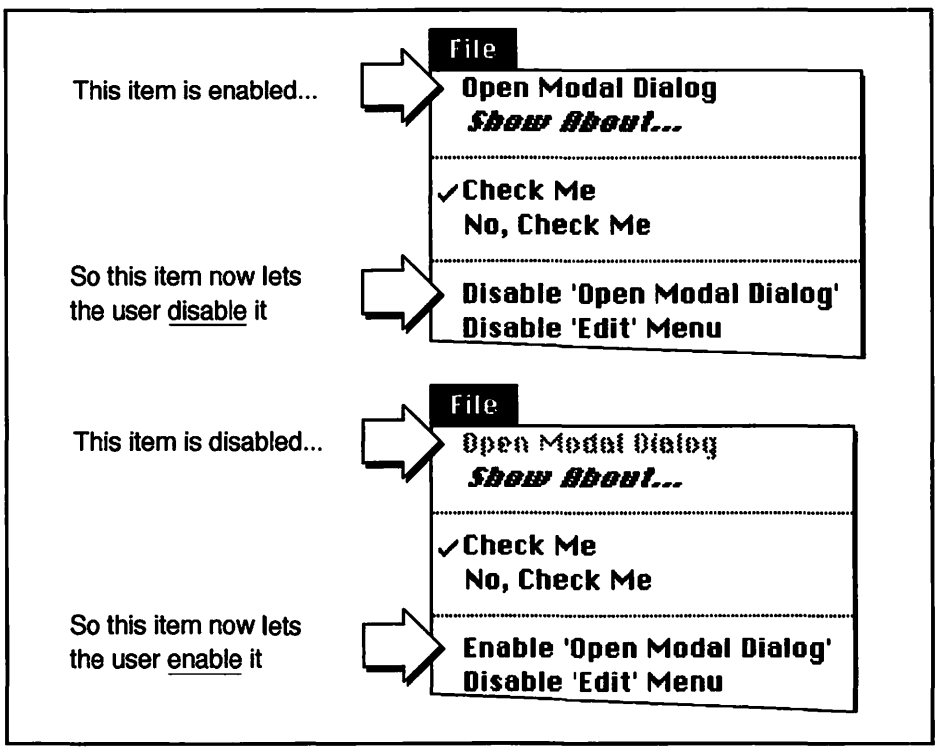


Figure 7-40. *Enabling/disabling an item and changing an item's text*

Handle_Disable_Edit_Item() works in the same way as the previous routine. The difference is in the second parameter passed to *EnableItem()*. By passing a value of 0 (*ENTIRE_MENU*) to *DisableItem()* you're telling the Toolbox to disable the entire Edit menu, not a particular item in it.

```
void  Handle_Disable_Edit_Item( void )
{
   if ( Edit_Menu_Disabled == TRUE )
   {
      EnableItem( Edit_Menu, ENTIRE_MENU );
      DrawMenuBar();
      SetItem( File_Menu, DISABLE_EDIT_MENU_ITEM, "\pDisable 'Edit' Menu");
      Edit_Menu_Disabled = FALSE;
   }
   else
   {
      DisableItem( Edit_Menu, ENTIRE_MENU );
      DrawMenuBar();
      SetItem( File_Menu, DISABLE_EDIT_MENU_ITEM, "\pEnable 'Edit' Menu");
      Edit_Menu_Disabled = TRUE;
   }
}
```

Handling a hierarchical menu

MenuMaster displays an alert if either of the hierarchical submenu items are selected. To display two different strings in the same alert the program uses the *ParamText()* trick discussed earlier.

```
void  Handle_Hierarchical_Menu( int the_item )
{
   switch ( the_item )
   {
      case SUBMENU_ITEM_1:
         ParamText("\pSubmenu, Item 1", "\p", "\p", "\p");
         NoteAlert( INFO_ALERT_ID, NIL );
         break;

      case SUBMENU_ITEM_2:
         ParamText("\pSubmenu, Item 2", "\p", "\p", "\p");
         NoteAlert( INFO_ALERT_ID, NIL );
         break;
   }
}
```

Chapter Summary

To display a menu bar in your Macintosh program you use 'MENU' resources and a single 'MBAR' resource. Each 'MENU' resource defines the menu items that appear in a single pull-down menu. The 'MBAR' resource packages the individual 'MENU' resources into single menu bar.

Several Toolbox routines are involved in setting up an application's menu bar. *GetNewMBar()* creates a menu list that holds a handle to each menu in the menu bar. *SetMenuBar()* installs the individual menus within the menu bar. *AddResMenu()* fills the Apple menu with the names of desk accessories and, under System 7, the names of items in the Apple Menu Items folder in the System Folder. Finally, the menu bar is displayed on the screen with a call to *DrawMenuBar()*.

To get access to a handle to an individual menu—a *MenuHandle*—call *GetMHandle()*. You'll then use this handle in subsequent calls to Toolbox routines that change the characteristics of the menu or items in it. Some of the changes you can make are: enabling and disabling a menu item, changing the name of a menu item, and displaying a checkmark by an item.

When the user clicks the mouse button, you'll want to check to see if the click took place in the menu bar area of the screen. A call to *FindWindow()* determines that. If the mouse down event did occur in the menu bar, you'll call the powerful Toolbox routine *MenuSelect()* to track the mouse in the menu bar, dropping down menus as the user moves the mouse over them.

If the user makes a selection from a menu, call *MenuSelect()* to determine what item was selected. You'll use the Toolbox routines *HiWord()* and *LoWord()* to extract both the menu and the menu item from the single value that *MenuSelect()* returns.

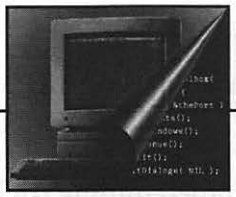
You can make things easier for the user by creating keyboard equivalents for commonly-used menu selections. You'll include the keyboard equivalent in the 'MENU' resource, then write a *Handle_Keystroke()* routine that keeps watch for this keystroke combination.

You can expand the amount of information in a menu by changing a menu item into a hierarchical menu. By marking a menu as such in the

'MENU' resource you'll add a pull-down menu to a menu item. You make your program aware of a hierarchical menu when you set up the program's menu bar. At that time you call *GetMenu()* and *InsertMenu()* for each hierarchical menu your program has.

To change menu characteristics you'll again rely on the Toolbox. *EnableItem()* and *DisableItem()* enable and disable a single menu item or an entire menu. Depending on the parameters you pass to it, the *CheckItem()* routine adds or takes away a checkmark from alongside a menu item. You can use *SetItemStyle()* to change the look of a menu item. You can give a menu item text characteristics such as bold or outline.

You can use a special filter function to give the user access to commands found in the Edit menu when a modal dialog is on the screen. Your program will call this filter function every time it calls the Toolbox routine *ModalDialog()*.



8 The Varying Mac

When the Macintosh was introduced a decade ago, there was just a single model. Now there are numerous models, each with a slightly different configuration. A user further complicates the picture by customizing a machine with a floating-point coprocessor, extra RAM, or a large-screen color monitor. The system software that drives the Macintosh has also evolved over the years.

While every Mac owner would like to have the most current, feature-laden model, the truth is that millions of Macintosh owners are running programs on older machines.

As a service to this varied audience, and to insure that your program has the widest distribution and usage as possible, you will want to write applications that execute on as many Macintosh models as possible.

Writing code that is guaranteed to run on several different models requires a little extra work on your part, but the effort will be worth it.

This chapter describes the programming tricks necessary to ensure that anyone using a Macintosh will also be able to use your applications easily.

Checking For Traps

The machine instructions for routines that you write exist, of course, within your compiled source code. The machine instructions for Toolbox routines, such as *DrawString()*, exist outside your compiled source code, this machine code is housed in ROM, or occasionally, in RAM. A Toolbox routine is also called a *trap*. The technique for placing shared system code outside of your compiled application is sometimes called *dynamic linking* or shared *libraries*. This is different from the library routines such as *strcpy()* that are compiled and linked together with your application code so that every application has its own copy of the compiled code.

If you are a Windows programmer, the Toolbox routines are similar to routines found in Windows DLLs (Dynamic Link Libraries).

Toolbox routines are traps

A Toolbox routine is usually located in ROM, though the System may on occasion load a routine in RAM. Where, exactly, is any one particular routine located in memory? The memory location of the routine is determined by the routine's *trap number*. If your application makes a call to *DrawString()*, the execution of your application will be interrupted while the processor makes use of the *DrawString()* trap number to locate the code for the *DrawString()* routine.

Earlier I said that Toolbox routines are in ROM. Why then am I now reneging and saying that some may be in RAM? For any given Macintosh, the contents of ROM are fixed. A new and improved ROM with additional routines may be included with the newer Macs. Can an older, existing model, with its older ROM, ever get these newer routines? Yes, when Apple provides a new System. The new System may contain patches—code that loads a routine from the System into RAM. This routine can be found in the ROM of a newer Mac and, via the patch, be placed in the RAM of an older Mac.

Traps are stored in RAM. Each trap number is associated with an address that is at the start of the code for the routine the trap represents. Figure 8–1 shows how a call to a hypothetical Toolbox routine called Routine_B results in the processor first going to a trap number in RAM—Trap #2. The address associated with Trap #2 is address_2. This address is the memory address of Routine_B. From there, the processor goes to address_2 in ROM to find and execute the code for the function Routine_B.

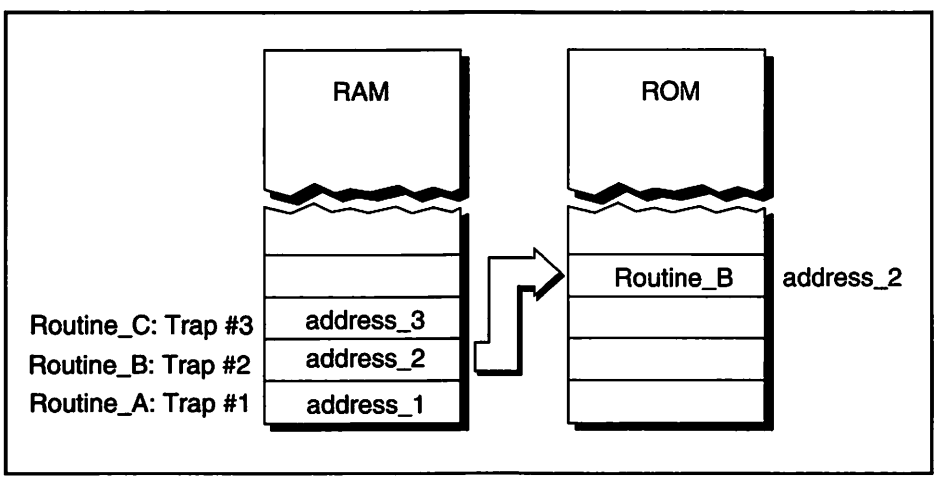


Figure 8–1. *A trap number leads to a Toolbox routine*

Let's sum up what you've learned to this point. Figure 8–2 shows what happens when your application makes a call to the Toolbox routine *DrawString()*.

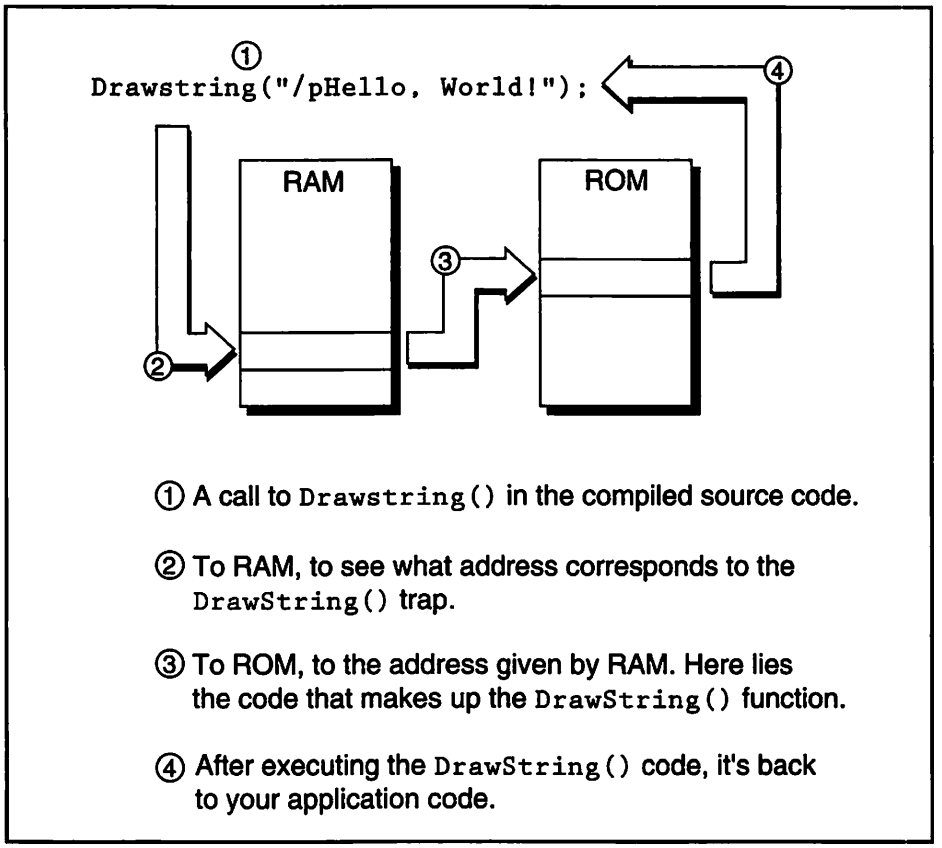


Figure 8–2. *Sequence of events in a call to* DrawString()

Each Toolbox routine is represented by a trap, and all of the traps are grouped together in RAM in a *dispatch table.* The dispatch table thus holds the starting address of each of the over two thousand Toolbox routines. For simplicity, imagine that the latest version of the Toolbox contains just three routines generically named Routine_A, Routine_B, and Routine_C. Figure 8–3 shows the dispatch table for a hypothetical Toolbox.

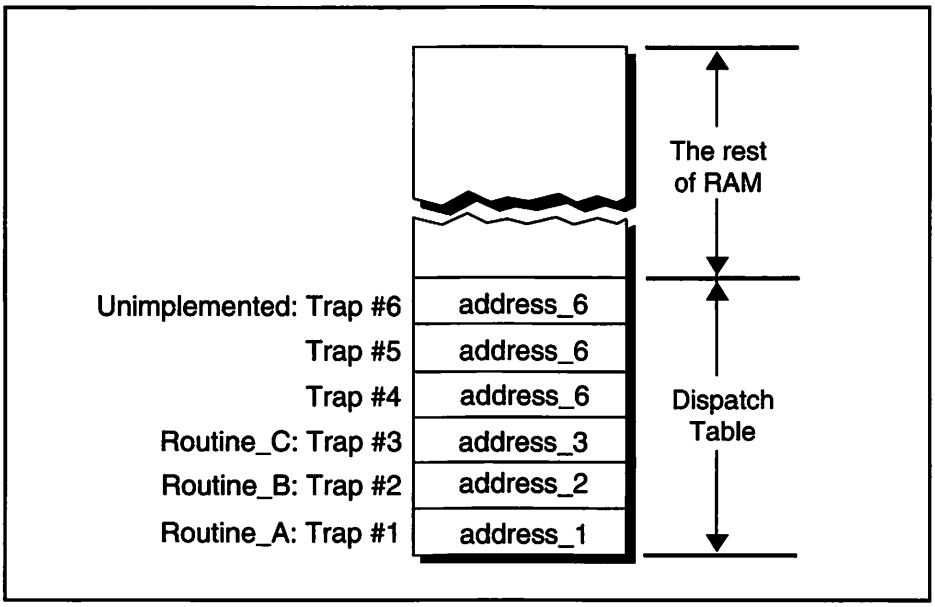


Figure 8–3. *A hypothetical dispatch table in RAM*

Figure 8–3 shows that a trap represents a Toolbox routine and has both a number and an address associated with it. Notice further that there are traps that don't represent any function—namely, Trap #4 and Trap #5. Notice further that these two empty traps both have the same address—address_6. This is the same address as the dispatch table entry labeled Unimplemented. Take careful note of this point; it will serve as the basis for determining whether a Toolbox routine exists in the Toolbox of the computer your application is running on.

The Unimplemented trap really does exist in our hypothetical Macintosh and in *every* real Macintosh computer. And Apple guarantees that it will never change its trap number and will never use it to house the address of a Toolbox routine.

A dispatch table is not full. As you just saw in Figure 8–3, it has empty entries. As Apple releases new versions of the System some of these previously empty entries will contain valid addresses that direct the processor to the code of new Toolbox routines. Figure 8–4 illustrates this.

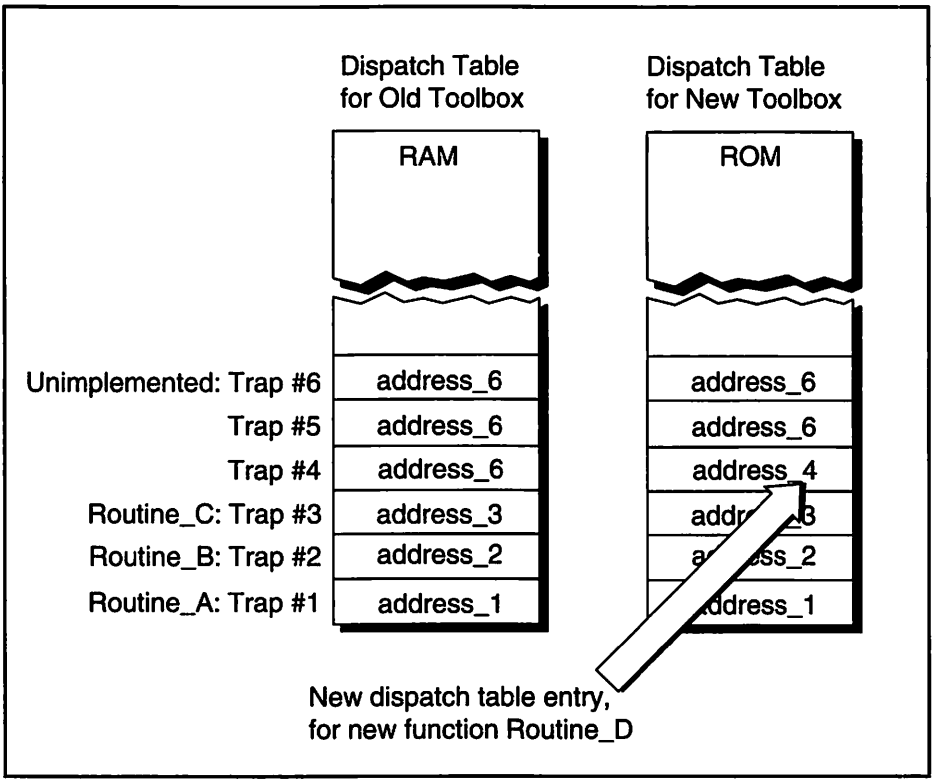


Figure 8–4. *The dispatch table for an old and a new Toolbox*

In Figure 8–4 the new dispatch table and the old dispatch table differ by just one entry. For the new Toolbox, Trap #4 now holds the address of a new Toolbox routine, Routine_D.

Now, after this very lengthy introduction to traps, I'm ready to cover the topic that is really of interest to you. Namely, if different Systems contain different versions of the Toolbox, how can you be sure that a Toolbox call you'd like to include in your source code is present on the Macintosh that will be running your application?

Why all the fuss about traps and their availability? Quite simply, if you attempt to make a call to a non-present Toolbox routine, your application will crash.

The answer to the above question lies in the fact that dispatch table entries that are empty all contain the identical address: the one found in the Unimplemented trap. Figure 8–5 demonstrates this.

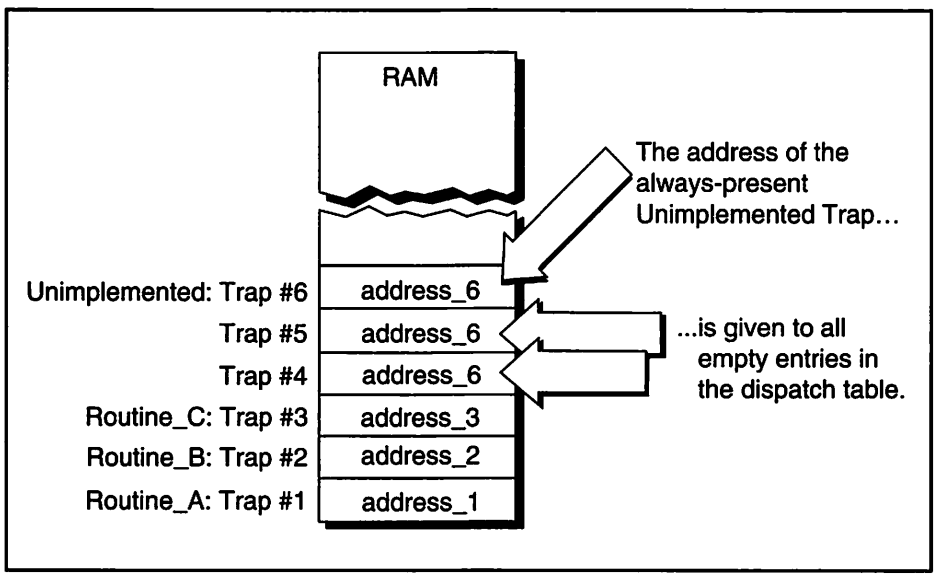


Figure 8–5. *Addresses in empty entries of the dispatch table*

To determine if a Toolbox routine is present you compare the address found in the trap number of the Toolbox routine to the address found in the trap number of the Unimplemented trap. Remember, empty entries have been assigned the same address as that placed in the Unimplemented trap. That means that if your comparison results in two addresses that are the same, the routine is *not* present in the version of the Toolbox you are checking. Figure 8–6 illustrates this, again using the hypothetical Toolbox.

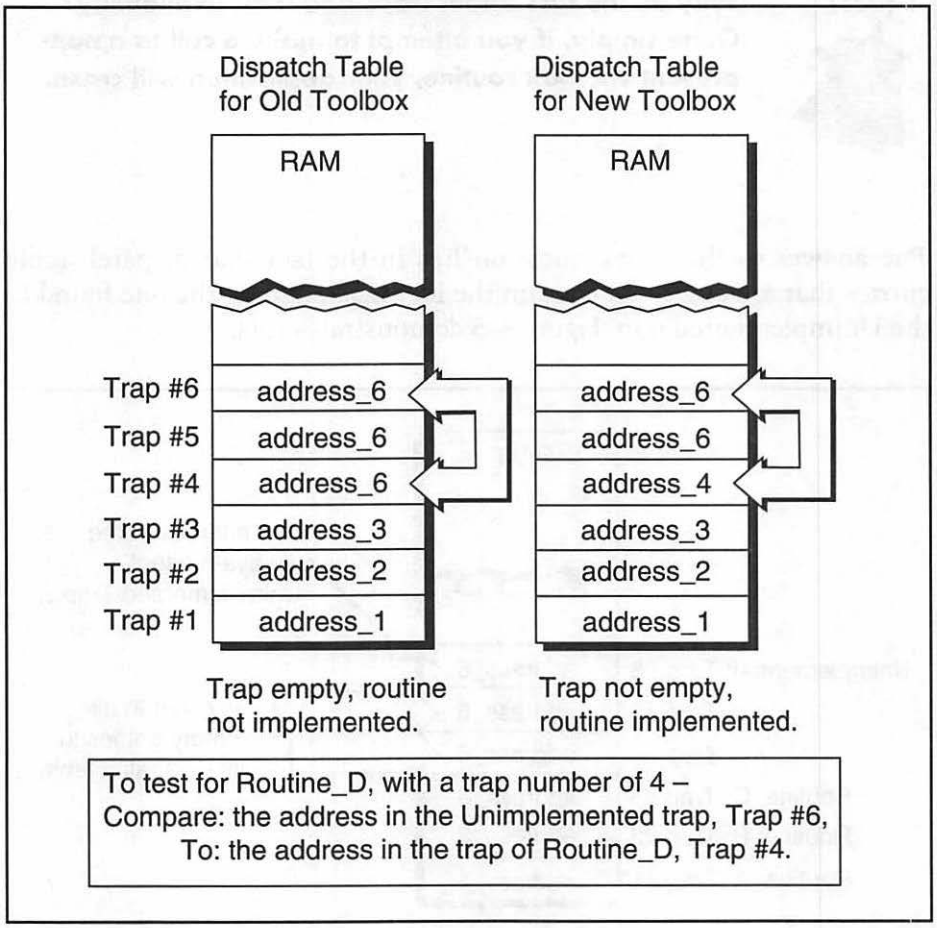


Figure 8-6. *Testing for routine Toolbox implementation*

Now that you know the theory behind checking for implemented Toolbox routines, it's time to move on to the real thing: the code to include in your application to perform this check.

Lesson 8-1: Traps and the Toolbox

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Determining if a Toolbox routine is implemented

In the previous section I said that each Toolbox routine has a trap number by which the code for the routine is accessed. There I used generic names and numbers for the traps. Here's a look at a C definition of the trap number for a Toolbox routine that loads a color window into memory, *GetNewCWindow()*:

```
#define   _GetNewCWindow   0xAA46
```

Compilers such as THINK C and Symantec C++ take care of definitions such as the above. You can include *_GetNewCWindow* and other traps in your program by including the *Traps.h* header at the top of your code:

```
#include <Traps.h>

main()
{
   ...
   ...
}
```

To refer to a trap in your source code, you simply preface the routine name with an underscore; it is not necessary to know the trap number. The trap number exists for the processor to use as an index into the dispatch table. Here's an example that uses *NGetTrapAddress()* to get the memory address of the *GetNewCWindow()* code, as found in the dispatch table:

```
long  color_wind_addr;

color_wind_addr = NGetTrapAddress(_GetNewCWindow, ToolTrap);
```

The above code by itself is not very useful. But when you also get the address of the Unimplemented trap, and then make a comparison of the two addresses, you have the solution to the problem of determining if a routine is present in the Toolbox. Below is a little C code used to check to see if the *GetNewCWindow()* function is in the Toolbox.

```
long  unimplemented_addr;

long  color_wind_addr;
```

```
unimplemented_addr = NGetTrapAddress(_Unimplemented, ToolTrap);
color_wind_addr    = NGetTrapAddress(_GetNewCWindow, ToolTrap);

if (color_wind_addr == unimplemented_addr)
{
   /* Trap is unavailable */
}
else
{
   /* Trap is available   */
}
```

The second parameter in the call to *NGetTrapAddress()—ToolTrap*—may have caught your eye. There is one final point to make about traps. There are actually two separate dispatch tables in RAM. One holds the traps for Operating System routines, while the other holds the traps for Toolbox routines. An example of an Operating System routine is *Eject()*, which, not surprisingly, ejects a disk from the disk drive. Examples of Toolbox routines are *MoveWindow()*, which moves a window, and the numerous drawing routines, such as *FrameRect()* and *PaintOval()*.

To allow you to distinguish between the Operating System traps and the Toolbox traps, Apple has created the Macintosh C enumerated type *TrapType*. There are two members to this type: *OSTrap* and *ToolTrap*.

In the previous code fragment, how would you know that *GetNewCWindow()* was a Toolbox trap and not an Operating System trap? One method is to look up the routine name in Apple's *Inside Macintosh* series of books. All trap numbers begin with $A. If the next digit in the trap number is between $0 and $7, then the trap is in the OS dispatch table and is of the *OSTrap* type. If the digit is instead between $8 and $F, then the trap is in the Toolbox dispatch table and is a *ToolTrap* type. You know that *GetNewCWindow()* is a *ToolTrap* type because the digit following the first $A (it too just happens to be a $A) falls in the range of $8 and $F. Figure 8–7 shows a listing of a few Toolbox trap numbers.

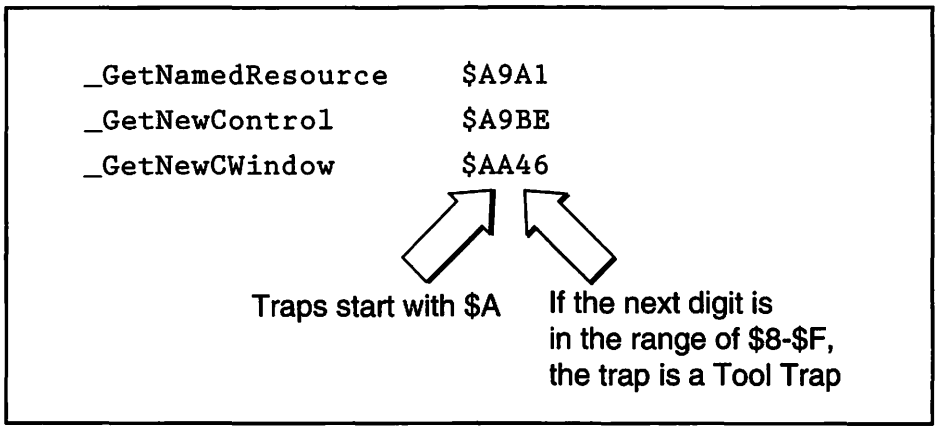


Figure 8-7. *Determining the type of a trap*

If you don't have a reference book handy there is a second method to determine a traps type. It's done by writing a couple of routines of your own, and it's a little tricky. By now you probably already know more about traps than you ever hoped you would. For that reason, I've thrown the routines into an appendix. If you're brave, or perhaps masochistic, refer to Appendix B.

So that you can fully understand what was transpiring, I intentionally made the previous code fragment a little wordy. Now that you are a trap master, I can tighten our C code up:

```
Boolean  Color_Wind_Available;

Color_Wind_Available = (NGetTrapAddress(_Unimplemented, ToolTrap)
                    != NGetTrapAddress(_GetNewCWindow, ToolTrap));

if (Color_Wind_Available)
{
   [ open a color window ]
}
else
{
   [ open a black and white window ]
}
```

In general, make your Boolean variable global and perform the trap check near the start of your code. Then you can use the *Boolean* every time you have to check for the presence of a Toolbox routine.

With the history of traps well established, earlier use of *NGetTrapAddress()* to determine if MultiFinder is present in the system should make sense to you. I introduced the technique but did not fully explain it back in Chapter 5. Now it should make better sense.

```
Boolean  Multifinder_Present;

Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                       NGetTrapAddress(_Unimplemented, ToolTrap));

if ( Multifinder_Present == TRUE )
   [ use the newer event-retrieving routine WaitNextEvent() ]
else
   [ use the older GetNextEvent() */
```

The Features of a Macintosh

The different members of the Macintosh family differ in the hardware features they contain. They can also differ in the version of System software they run. To make matters worse (for you, the compatibility-minded programmer), users can make a host of changes to both the hardware and software once they get their computers home or to the office.

The previous section demonstrated a means of determining if a particular Toolbox routine is present through the use of the *NGetTrapAddress()* routine. There will also be times when you want to know if the computer your program is running on has a particular hardware feature. One way to get this information is by making a call to the Toolbox routine *SysEnvirons()*. Here's a typical call:

```
SysEnvRec  mac_info;

SysEnvirons(curSysEnvVers, &mac_info);
```

As Macintosh features have evolved, so has *SysEnvirons()*—there is more than one version available. To make use of *SysEnvirons()* you pass it the version you'll be using. Always pass *curSysEnvVers* as the version number. Your compiler defines this constant for you; you needn't worry about its value.

The second parameter to pass is a pointer to a variable of type *SysEnvRec*. After making the call, the several members of the *SysEnvRec* structure will yield useful information such as the CPU the machine has and the version of the System software currently running. Below is the *SysEnvRec* structure.

```
struct SysEnvRec
{
   short    environsVersion;
   short    machineType;
   short    systemVersion;
   short    processor;
   Boolean  hasFPU;
   Boolean  hasColorQD;
   short    keyBoardType;
   short    atDrvrVersNum;
   short    sysVRefNum;
};
```

As this section ends, you'll notice that I haven't provided you with an in-depth example of *SysEnvirons()*. The next section tells you why.

More Features of a Mac

Starting with the release of System 6.0.4 back in 1989, the use of *SysEnvirons()* became virtually obsolete. After reading the material in the last section, you're probably wondering why I didn't mention this fact a little sooner!

The *Gestalt()* function

All the information about machine features is useful stuff, even if the calls to *SysEnvirons()* have lost much of their value. The way in which the Macintosh determines the features of a machine is different than you've seen for other brands of computers. It is important that you understand these concepts. And there will be some occasions when you'll want to call *SysEnvirons()*. So, be truthful now, if I had told you at the onset of the last section that the primary routine I'd be covering was virtually obsolete, would you really have read it?

System 6.0.4 introduced a new Toolbox routine, *Gestalt()*. This function does what *SysEnvirons()* did, and much more. When it comes to determining the various features on a Macintosh, *SysEnvirons()* pales in comparison to *Gestalt()*. Figure 8–8 sums this up.

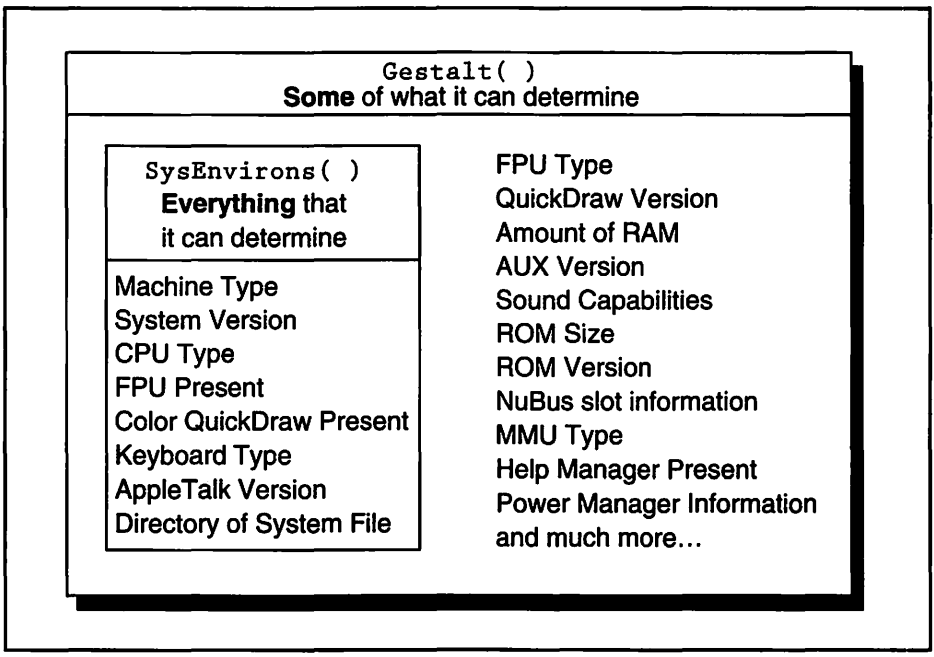


Figure 8–8. *The advantages of* Gestalt()

Now that you know about the existence of the amazing *Gestalt()* function, why would you ever bother with *SysEnvirons()*? Because there's a catch to *Gestalt()*; it's only available on more recent machines. If it is available, you'll want to use it. If not, you'll have to use the older *SysEnvirons()* routine.

Checking for the availability of *Gestalt()*

The *Gestalt()* function is available on Macs running System 6.0.4 and later, including any version of System 7. Since System 6.0.4 was released in 1989, most Mac owners have a version at least that new on their Macs. That means there's a very good chance that any Mac your application runs on will support *Gestalt()*. But you, of course, can't make that assumption.

Life is hard enough for those unfortunate enough to still be working on pre-1989 Macs, don't aggravate them by crashing their machines! Instead, make a couple of checks very early in your program to see just what computer your program is running on. Because you won't immediately know if *Gestalt()* is available, use *SysEnvirons()*. Here's an example:

```
SysEnvRec  mac_info;

SysEnvirons(curSysEnvVers, &mac_info);

if (mac_info.machineType < 0)
   ExitToShell();

if (mac_info.systemVersion < 0x0604)
   ExitToShell();
```

The *SysEnvirons()* routine was added to the Toolbox back in 1986, so it's a pretty safe bet that it's available on any Mac your program will see. All right, you caught me—I'm guilty of making an assumption! Here's one more: anyone still using a pre-1986 Macintosh won't possibly be interested in the amazing, state-of-the-art application you'll be writing anyway!

Now, back to work. After making the call to *SysEnviron()* you examine two of the members of the *SysEnvRec* structure, the *machineType* and the *systemVersion*.

A *machineType* of 0 means that this Mac is really old, so you'll use the routine *ExitToShell()* to quickly exit your program and return the user to the Finder.

The same applies to a system of less than 0x0604. This hexadecimal value means that the System software is pre-6.0.4 and thus *pre-Gestalt()*. I'll have more to say about the hexadecimal display of the system version in just a few pages.

You'll be using *Gestalt()* to check for some Mac features, so you want to establish that the Mac contains a System with the *Gestalt()* function. If it doesn't, again use *ExitToShell()* to terminate your program.

It's a good idea to exit a program in a more graceful manner then I just did in the above example. Before you ever abnormally terminate a program you will want to give the user some information as to why he's being

whisked back to the Finder. This information comes in the form of an alert that displays an informative message. Here I've rewritten the previous example to include the display of an alert. This chapter's example program will do the same. Chapter 9 covers error-handling even more extensively.

```
#define    TOO_OLD_ALRT_ID     129
#define    NIL                  0L

SysEnvRec  mac_info;

SysEnvirons( curSysEnvVers, &mac_info );

if ((mac_info.machineType < 0) || (mac_info.systemVersion < 0x0604))
{
   StopAlert( TOO_OLD_ALRT_ID, NIL );
   ExitToShell();
}
```

The technique used here handles the case of users with a pre-1989 System on their Mac by exiting the program. If you want to write a program that can execute on a Mac with an old system, you can't use the *Gestalt()* function—it didn't exist then. If your application doesn't include any code that makes any assumptions about the Macintosh it's running on, that won't be a problem. Then you'll skip the *systemVersion* test.

Once you've made it past these two checks, you know that your application is running on a Macintosh that supports *Gestalt()*, and you can freely use *Gestalt()* anywhere in your program.

Determining machine features using *Gestalt()*

To use *Gestalt()*, you pass it a *selector code* that tells *Gestalt()* what hardware or software feature you want to examine.

In return, *Gestalt()* returns a *response parameter*. The response parameter is the answer to the question you posed in the selector code. Here's an example that checks for the version of QuickDraw in a Mac.

```
#include <GestaltEqu.h>

OSErr  err;
long   response;

err = Gestalt(gestaltQuickdrawVersion, &response);
```

In this example *gestaltQuickdrawVersion* is the selector code and response is the response parameter. This call asks *Gestalt()* to return the version of QuickDraw in the machine. After the call to *Gestalt()* is complete response will have one of the following values:

```
gestaltOriginalQD
gestalt8BitQD
gestalt32BitQD
gestalt32BitQD11
gestalt32BitQD12
gestalt32BitQD13
```

You can use the above as constants because your compiler defines them in an enumerated type. This enum is in a header file called *GestaltEqu.h*. Here are the actual values from part of that enum:

```
gestaltOriginalQD = 0x000,   /* original 1-bit QD   */
gestalt8BitQD     = 0x100,   /* 8-bit color QD      */
gestalt32BitQD    = 0x200,   /* 32-bit color QD     */
gestalt32BitQD11  = 0x210,   /* 32-bit color QDv1.1 */
gestalt32BitQD12  = 0x220,   /* 32-bit color QDv1.2 */
gestalt32BitQD13  = 0x230,   /* 32-bit color QDv1.3 */
```

The constant *gestaltQuickdrawVersion*, along with numerous other selector codes, is also defined in the *GestaltEqu.h* header file. If you use *Gestalt()* in your program you must use *GestaltEqu.h* in an *#include* at the start of your program. Appendix C shows many of the selector codes and responses included in this file.

Gestalt() gives you verification that it was able to return the requested information in the form of a result code of type *OSErr*. After a call to *Gestalt()* always compare the result code to *noErr*. Your compiler definition for *noErr* looks like this:

```
#define  noErr  0
```

If *Gestalt()* returns a result code of 0 the call was successful. If it's any other value you should not base the code that follows on the response that *Gestalt()* returned. The following example shows a call to *Gestalt()* and a test of the returned result code.

```
#include <GestaltEqu.h>

OSErr  err;
long   response;

err = Gestalt(gestaltQuickdrawVersion, &response);

if ( err == noErr )
{
   if ( response == gestaltOriginalQD )
      DrawString("\pYou have the original version of QuickDraw.");
}
else
   DrawString("\pGestalt error.");
```

Figure 8–9 sums up a call to *Gestalt()*.

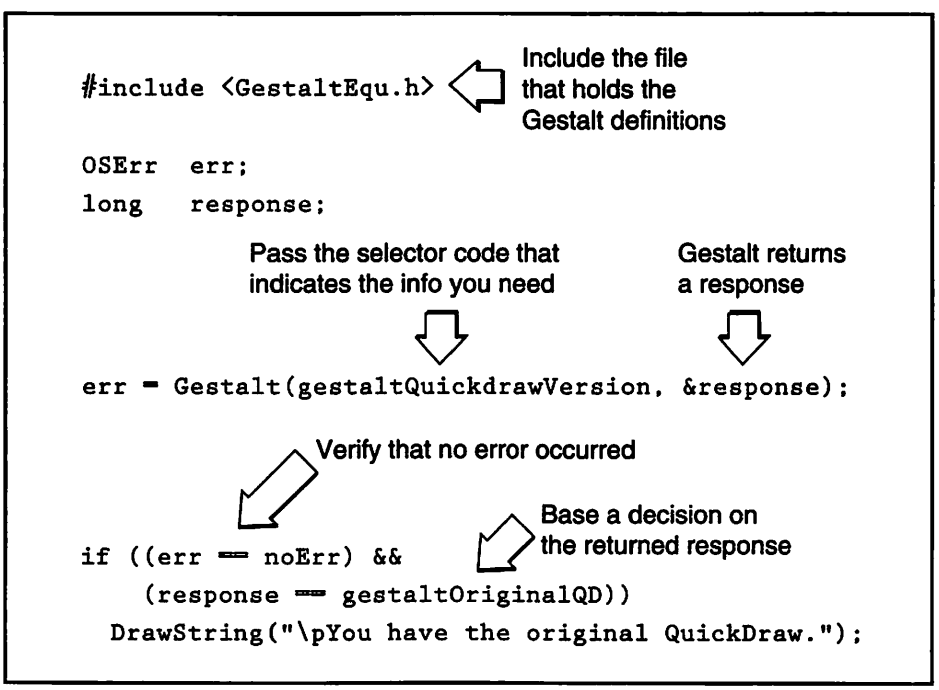


Figure 8–9. *Using* Gestalt()

Now that you know just how to use *Gestalt()*, what can you use it for? The following sections cover a few of the Macintosh features you can determine with *Gestalt()*. Appendix C covers several more.

Determining the QuickDraw version

The drawing routines that make up QuickDraw have been improved and increased over the years. The original version did not support color; subsequent versions do.

If you're going to work with color, the Mac your program runs on must have a version of QuickDraw that supports color. Use the selector *gestaltQuickdrawVersion* to determine the version of QuickDraw that is currently present.

```
Selector code
gestaltQuickdrawVersion       /*  QuickDraw version     */
Response parameter
gestaltOriginalQD = 0x000     /*  original 1-bit QD     */
gestalt8BitQD     = 0x100     /*  8-bit color QD        */
gestalt32BitQD    = 0x200     /*  32-bit color QD       */
gestalt32BitQD11  = 0x210     /*  32-bit color QDv1.1   */
gestalt32BitQD12  = 0x220     /*  32-bit color QDv1.2   */
gestalt32BitQD13  = 0x230     /*  32-bit color QDv1.3   */
Example
Boolean  Color_QD_Present;
OSErr    err;
long     response;

err = Gestalt(gestaltQuickdrawVersion, &response);

if ( err == noErr )
{
   if ( response == gestaltOriginalQD )
      Color_QD_Present = FALSE;
   else
      Color_QD_Present = TRUE;
}
else
   DrawString("\pGestalt error.");
```

Determining the CPU type

All Macintoshes use a CPU, or central processing unit, from the Motorola 680x0 family. The oldest, the 68000, was in the original Macintosh. Use the *gestaltProcessorType* selector to determine which CPU is in the Macintosh your application is running on.

```
Selector code
gestaltProcessorType     /*  processor type     */
Response parameter
gestalt68000 = 1
gestalt68010 = 2
gestalt68020 = 3
gestalt68030 = 4
gestalt68040 = 5
Example
OSErr  err;
long   response;

err = Gestalt(gestaltProcessorType, &response);

if ( err == noErr )
{
   if ( response == gestalt68040)
      DrawString("\pYou have the newest CPU available!");
   else
      DrawString("\pNo 68040? Tempted to upgrade?");
}
else
   DrawString("\pGestalt error.");
```

Determining the amount of physical RAM

In the last few years RAM prices have dropped considerably. Consequently, many Macs have plenty of RAM. You may be fortunate enough to have a Mac loaded with RAM, but compatibility concerns dictate that you keep in mind the less fortunate! Millions of Macs with 1 Mb of RAM were sold, and many are still in daily use.

If you want to check for the amount of RAM a machine has, use the *gestaltPhysicalRAMSize* selector in a call to *Gestalt()*. The response will be the number of bytes of physical RAM.

```
Selector code
gestaltPhysicalRAMSize      /*  physical RAM size  */
Response parameter
Number of bytes of RAM.
Example
#define  ONE_K   1024.0

OSErr    err;
long     response;
short    mega_bytes;
short    k_bytes;

err = Gestalt(gestaltPhysicalRAMSize, &response);

if (err == noErr)
{
   mega_bytes = response/(ONE_K * ONE_K);  /*  convert bytes */
   k_bytes    = response/ONE_K;            /*  to Mb and K   */
}
```

Determining the floating-point coprocessor type

Some Macs have a floating-point coprocessor installed. Using the selector *gestaltFPUType, Gestalt()* will return a value that indicates the type of floating-point coprocessor installed, if any.

```
Selector code
gestaltFPUType           /*  FPU type           */
Response parameter
gestaltNoFPU     = 0     /*  no FPU present     */
gestalt68881     = 1     /*  68881 FPU          */
gestalt68882     = 2     /*  68882 FPU          */
gestalt68040FPU = 3      /*  68040 built-in FPU */
Example
OSErr  err;
long   response;

err = Gestalt(gestaltFPUType, &response);

if ( err == noErr )
{
   if ( response > gestaltNoFPU )
      /* This machine has a floating-point unit */
   else
```

```
        /* No floating-point unit present          */
    }
    else
       DrawString("\pGestalt error.");
```

Determining the Macintosh machine type

You can determine the type of Macintosh, or machine, your application is running on by passing the *gestaltMachineType* selector to *Gestalt()*. But be aware that two Macs of the same type may be running different systems, have different amounts of memory, or differ in other ways. Because they may differ in many respects you should not use the machine type to assume certain features do or don't exist on the user's computer.

```
Selector code
gestaltMachineType        /*  machine type            */
Response parameter
kMachineNameStrID    = -16395
gestaltClassic       = 1
gestaltMacXL         = 2
gestaltMac512KE      = 3
gestaltMacPlus       = 4
gestaltMacSE         = 5
gestaltMacII         = 6
gestaltMacIIx        = 7
gestaltMacIIcx       = 8
gestaltMacSE030      = 9
gestaltPortable      = 10
gestaltMacIIci       = 11
gestaltMacIIfx       = 13
gestaltMacClassic    = 17
gestaltMacIIsi       = 18
gestaltMacLC         = 19
gestaltQuadra900     = 20
gestaltPowerBook170  = 21
gestaltQuadra700     = 22
gestaltClassicII     = 23
gestaltPowerBook100  = 24
gestaltPowerBook140  = 25
Example
OSErr  err;
long   response;
```

```
err = Gestalt(gestaltMachineType, &response);

if (err == noErr)
{
   DrawString("\My senses tell me you're using a...");

   switch (response)
   {
      case gestaltMacPlus:
         DrawString("\pMacintosh Plus!");
         break;
      case gestaltMacSE:
         DrawString("\pMacintosh SE!");
         break;

      [ use "cases" for any or all Mac types ]

      default:
         DrawString("\pI give up!");
         break;
   }
}
else
   DrawString("\pGestalt error.");
```

Determining the operating system version

The operating system version number can be determined by using the *gestaltSystemVersion* selector. Like the machine type, knowledge of the operating system version does not lend enough information to make programming decisions regarding the features of a particular Macintosh model.

The response that *Gestalt()* returns is a hexadecimal representation of the system version. For example, if the system is version 6.0.4, *response* will be 0x0604. If the system version is 7.1.0, *response* will be 0x0710.

```
Selector code
gestaltSystemVersion          /*  System version    */
Response parameter
Number of the System file, in hexadecimal form.
Example
Boolean  System_7_Present;
```

```
OSErr    err;
long     response;

err = Gestalt(gestaltSystemVersion, &response);

if ( err == noErr )
{
   if ( response >= 0x0700 )
      System_7_Present = TRUE;
   else
      System_7_Present = FALSE;
}
else
   DrawString("\pGestalt error.");
```

Monitor-Aware

The original Ford Model T car came in a choice of colors: black, or black. Like the car, the original Macintosh model came with your choice of monitor: built-in 9-inch diagonal black and white or...you get the point. No choice of size, no choice of color display. Things have changed in ten years, and so have the tricks you'll need to use to make sure the programs you write are compatible with both color and monochrome monitors, with monitors of different sizes, and with Mac systems with more than one monitor. We'll cover multiple monitors first.

Dealing with multiple monitors

Though most users have just a single monitor, don't assume this is so. If you don't allow the user to drag windows across monitors, or a window comes up centered between two monitors, the user will quickly become frustrated with your program.

Setting the window drag region

In Chapter 5 I used the boundaries of the screen to set the boundaries for dragging a window. That method works just fine for a system that has a single monitor. If you don't want to make that assumption (and you shouldn't) you'll need to use a reference other than the screen boundaries.

For a system with multiple monitors, one of your main concerns is that you properly set the boundaries for window dragging. The size of this drag boundary rectangle will be dependent on the size of the monitor. More correctly, it will be dependent on the area that makes up the desktop. Because the desktop is usually gray, this area is known as the *gray region*. Formally, it consists of a region that is the union of any active screen devices (monitors) minus the menu bar. Figure 8–10 shows the gray region for a dual-monitor system.

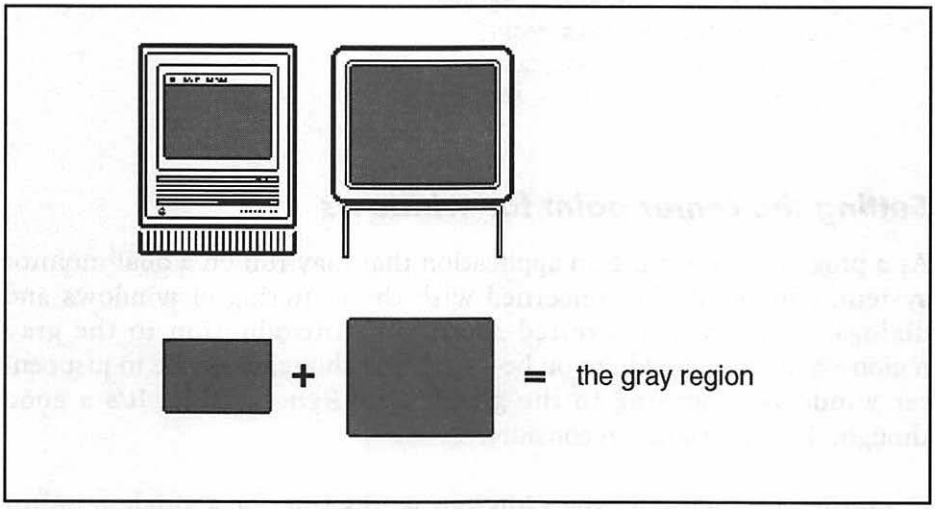


Figure 8–10. *The gray region of a dual-monitor system*

If you want to give the user the ability to drag windows created by your application across monitors, you need to set up your drag boundary rectangle so that it encompasses the entire gray region. Luckily, a routine you call during initialization does much of the work for you. When your initialization routine calls *InitWindows()* it calculates this region and saves it to a global *rgnHandle* variable called *GrayRgn*. Because it's a system global, you can use the *GrayRgn* variable without declaring it in your programs.

The following is a replacement for the *Set_Window_Drag_Boundaries()* function created in Chapter 5. It creates a rectangle independent of the number of monitors running—a drag rectangle set to the size of the gray region. The rest of the routine is the same as the old version. I inset the drag rectangle a few pixels so that the drag region is not quite as big as

the gray region, thus preventing the user from dragging windows off the screen. Here's the new routine:

```
#define   DRAG_EDGE   10

Rect  Drag_Rect;

void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = (**(GrayRgn)).rgnBBox;
   Drag_Rect.left += DRAG_EDGE;
   Drag_Rect.right -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}
```

Setting the center point for windows

As a programmer writing an application that may run on a dual-monitor system, you should be concerned with the centering of windows and dialogs. If you're still excited about your introduction to the gray region—and why wouldn't you be—your first thought may be to just center windows according to the global *GrayRgn* variable. It's a good thought, but you should reconsider.

Centering a window by the *GrayRgn* works fine for a single-monitor system. For a dual-monitor Mac it would place the window between the two monitors, something that would definitely be disadvantageous unusable for the user. Figure 8–11 illustrates the results of window centering using *GrayRgn*.

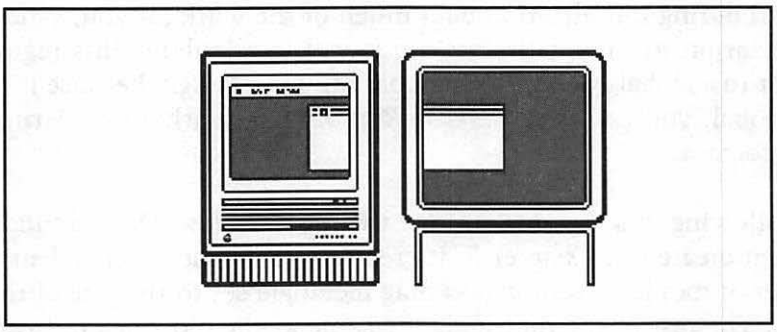


Figure 8–11. *Improper window centering on a dual-monitor system*

Instead of using the entire desktop for centering, as you do for window dragging, you can use just the *main screen*: the screen that displays the menu bar. The window or dialog you're going to bring to the screen and center is no doubt a result of the user choosing a menu option. So it is most likely that's where the user is focused—on the monitor with the menu bar.

To center the window you need to get the gray area of that one monitor. That's a trick that you can accomplish with a call to the routine *GetMainDevice()*.

A monitor is a graphic display *device*. When the Mac starts up it checks its expansion slots for display devices. The Mac stores the information it obtains for a device in a device structure—a *GDevice* structure to be exact. It then stores these structures in a *device list*. I'll talk more about devices when I discuss color issues. For now, I'm only interested in getting a handle to the main display device; that is, the device that displays the menu bar. That's exactly what *GetMainDevice()* does.

One of the members of a *GDevice* structure is *gdRect*. This rectangle is the boundary rectangle of the device's display. Figure 8–12 shows the relationship between *GDHandle* and *GDevice*. For simplicity, just one of the members of the *GDevice* structure is shown.

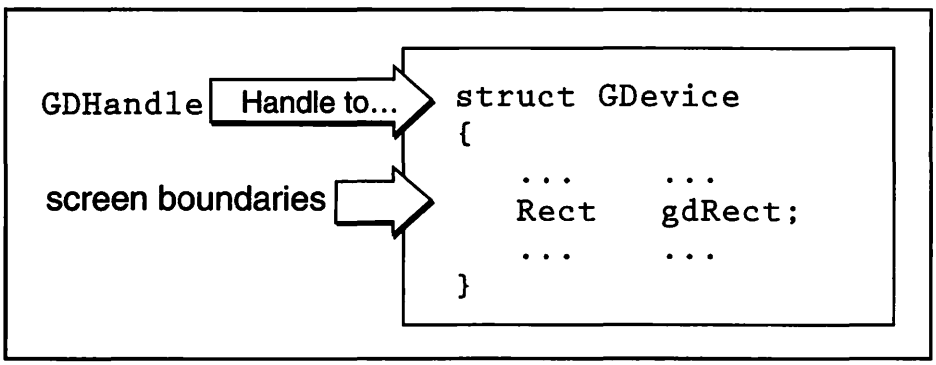


Figure 8–12. GDHandle *is a handle to a* GDevice *structure*

The boundary rectangle includes both the gray area and the area of the menu bar. You can thus use the bounds of *gdRect* to determine the center of the main screen. Here's how:

```
#define   MENU_BAR_HEIGHT   18

Point  Screen_Center;

void  Set_Screen_Center( void )
{
   GDHandle  gd_handle;
   Rect      bnds_rect;

   gd_handle = GetMainDevice();

   bnds_rect = (**(gd_handle)).gdRect;

   Screen_Center.h = (bnds_rect.right /2);
   Screen_Center.v = (bnds_rect.bottom/2) + (MENU_BAR_HEIGHT/2);
}
```

The *Set_Screen_Center()* routine calls *GetMainDevice()* to get a handle to the display device holding the main screen. Dereference the handle so that you can look at this screen's boundary rectangle, *gdRect*. Make the *Point* variable that holds the center point, *Screen_Center*, global so that you can use it throughout your program for centering anything you want.

Data structures and routines that Apple introduced to support the use of graphics devices are a part of Color QuickDraw. If the user doesn't have Color QuickDraw, you can't use them.

A second method for centering a window on the main screen is to use a QuickDraw global variable called *screenBits*. This variable is a structure that represents a bit map of the main screen. The bounds member is a *Rect* that defines the screen of the main display. Here, in its entirety, is a means to determine the center of the main screen without using graphics device structures:

```
Screen_Center.h = screenBits.bounds.right/2;
Screen_Center.v = (screenBits.bounds.bottom/2) +
((MENU_BAR_HEIGHT/2)/2);
```

Easy, huh? So why did I go through the much longer explanation using the *GDHandle*? Because you'll need all of this information on graphics devices for the upcoming discussion about working with color. Determining the screen center from the *GDHandle* provides you with a sound explanation of device theory.

Once you know the center of the screen it's simple math to center a window or dialog. I use *GetNewWindow()* to demonstrate how to center a window using the *Screen_Center* point.

```
#define   WIND_WIDTH    500
#define   WIND_HEIGHT   300

Point      Screen_Center;
WindowPtr  The_Window;

void  Open_Window( void )
{
   short  top, left;

   The_Window = GetNewWindow(400, 0L, (WindowPtr)-1L);

   if ( The_Window == NIL )
      ExitToShell();

   left = Screen_Center.h - (WIND_WIDTH /2);
   top  = Screen_Center.v - (WIND_HEIGHT/2);

   MoveWindow(The_Window, left, top, TRUE);

   ShowWindow(The_Window);
}
```

After *GetNewWindow()* loads a window into memory, use the *Point* variable *Screen_Center* to establish the top left corner of the window. You set the width and height of the window when you create the 'WIND' resource in ResEdit. Use *MoveWindow()* to move the window to the center of the screen.

If the 'WIND' resource that defines this window made the window invisible, then the centering of the window took place behind the scenes. Now it's time to display it with a call to *ShowWindow()*.

Dealing with different sized monitors

In the previous section you saw how to use the *GrayRgn* global variable to determine the boundaries of the desktop for a system that has more than one monitor. You then learned how to make a call to *GetMainDevice()* to determine the center of the screen that holds the menu bar. Both of these techniques, used to avoid problems should multiple monitors be present, work for a single monitor system regardless of the screen size of the monitor.

You should use last section's *Set_Window_Drag_Boundaries()* routine in all your applications; it works for single- or dual-monitor systems, regardless of the size of the monitor. The same is true for the *Set_Screen_Center()* routine and the window centering technique used in the *Open_Window()* example.

Color Aware

The way in which your program behaves may be dependent on the monitor on which the user displays your program. To make your program truly compatible with the variety of Macintosh systems on the market, you'll want it to be able to display color on a color Macintosh while still being able to run on a monochrome system.

Color representation

A monochrome monitor represents a single pixel on the screen by a single bit of memory. A bit has two possible values, 0 and 1, so any pixel on the screen of a monochrome monitor can have two possible values: white or black.

To allow a pixel to be capable of displaying more than two colors, that pixel must be represented by more than a single bit of memory. If two bits are used per pixel, then a pixel can take on any one of four colors. Four bits per pixel yields 16 colors, while eight bits gives 256 colors. Using eight bits (a byte) of memory per pixel is common. After eight bits comes 16-bit and 24-bit color representation. These are usually reserved for high-end, expensive systems.

The number of bits that represent a single pixel is the monitor's *pixel depth*, or pixel value. Determining a monitor's pixel depth will be the primary focus of this section.

Knowing the pixel depth of the monitor that is displaying your program is important because your program will make decisions based on the level of color the monitor can display. Here's a typical decision your program might make:

```
if machine has color, and monitor is set to display it
   draw color text
otherwise
   draw black and white text
```

Another example is the displaying of pictures. If you're going to display a picture in a window you might want to have two or three separate ones to pick from. The one you choose will depend on the amount of color the user's Macintosh can display. When a black and white Mac shows a color picture the computer translates the colors to black and white. The Mac displays similar shades of a dark color as black. If these colors are adjacent, the areas that should be separate and distinct will blend into one. Figure 8–13 shows the display of the same picture on both a four-color monitor and a monochrome monitor. Note that the monochrome monitor can't make the distinction between shades and produces an undesirable display of your picture.

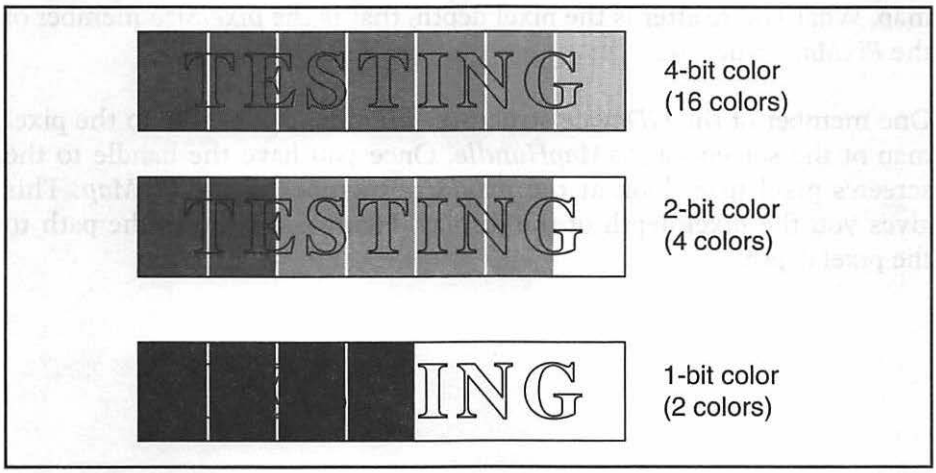


Figure 8–13. *The same picture viewed at 4-, 2-, and 1-bit color*

Getting the pixel depth of a monitor

When I discussed how to determine the center of a monitor I introduced the ideas of *graphic devices* and the *device list*. A monitor is a graphic device. More technically, the monitor's video card is a graphic device. The RAM memory that holds the value of each display pixel is located on the video card, not the RAM in the Macintosh. Information about a graphic device is stored in a *GDevice* structure, which is in turn placed in a device list. Recall that a *GDHandle* is a handle to a *GDevice*.

You obtain a handle to the first device in the device list by calling *GetDeviceList()*. Once you have the handle, pass it to a routine that determines the pixel depth of the display:

```
GDHandle  current_device;
short     pixel_depth;

current_device = GetDeviceList();
pixel_depth = Get_Pixel_Depth(current_device);
```

Before you look at *Get_Pixel_Depth()*, a little background information is necessary.

A graphic display device has a *PixMap*—a pixel map that defines the display. A *PixMap* is a structure that holds such information as the starting address of the device's video RAM and the depth of each pixel in the map. What you're after is the pixel depth; that is the *pixelSize* member of the *PixMap* structure.

One member of the *GDevice* structure, *gdPMap*, is a handle to the pixel map of the screen—a *PixMapHandle*. Once you have the handle to the screen's pixel map, look at the *pixelSize* member of the *PixMap*. This gives you the pixel depth of the display. Figure 8–14 shows the path to the pixel depth.

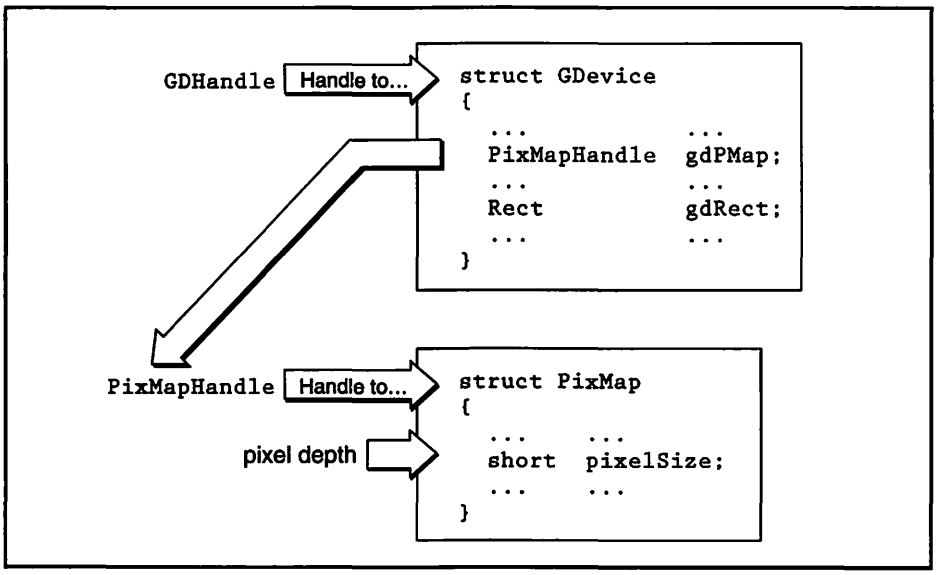


Figure 8-14. *The path from* GDHandle *to* pixelSize

Now that you have background information on *PixMaps*, *Get_Pixel_Depth()* should make sense to you. Here it is.

```
short  Get_Pixel_Depth(GDHandle  the_device)
{
   PixMapHandle  screen_PixMap_handle;
   short         pixel_depth;

   screen_PixMap_handle = (**(the_device)).gdPMap;
   pixel_depth = (**(screen_PixMap_handle)).pixelSize;
   return  pixel_depth;
}
```

Get_Pixel_Depth() first takes the passed in *GDHandle* variable and dereferences it get to the *GDevice* structure. There you get *gdPMap*, a *PixMap* handle to the screen of the device being examined. Because you have a handle, not the actual pixel map itself, you have to dereference it to get to the *pixelSize* member of the *PixMap*.

Now that you have the pixel depth of a monitor, how do you use it? One use is for displaying pictures to a window. Your technique might be to store two pictures in a resource file—one designed for monochrome systems and one for color systems, as shown in Figure 8–15. Then when you determine the pixel depth of the system your program is running on, you display the appropriate picture in a window.

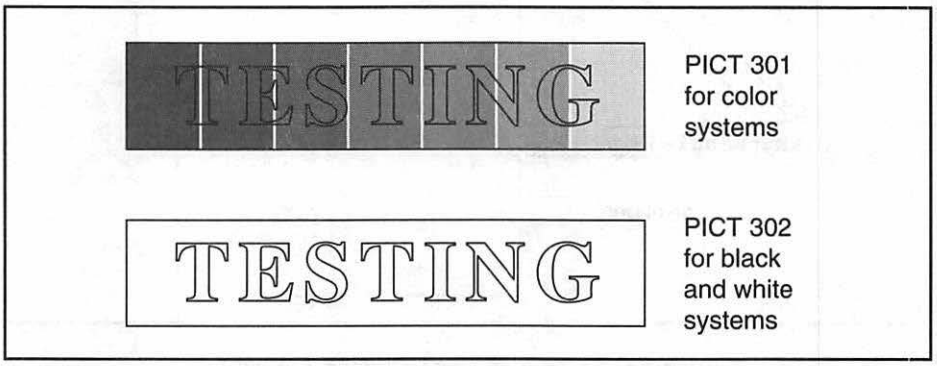
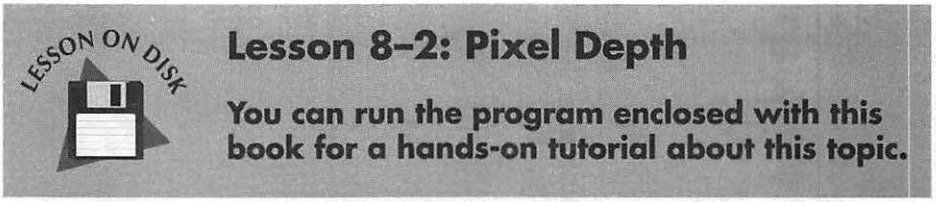


Figure 8–15. *Two similar PICTs: one for color, one for monochrome*



Multiple-monitors and pixel depth

What about a system that has more than one monitor? If there is more than one display device, your interest will be in determining the pixel depth of each monitor and saving the minimum depth. Why? If one monitor is black and white, and a second is color, you'll want to display program features in monochrome so that they are properly viewed on both monitors. If you displayed 'PICT' 301 from Figure 8–15 in a window that spanned both a color and monochrome monitor, the result would be undesirable, as demonstrated in Figure 8–16.

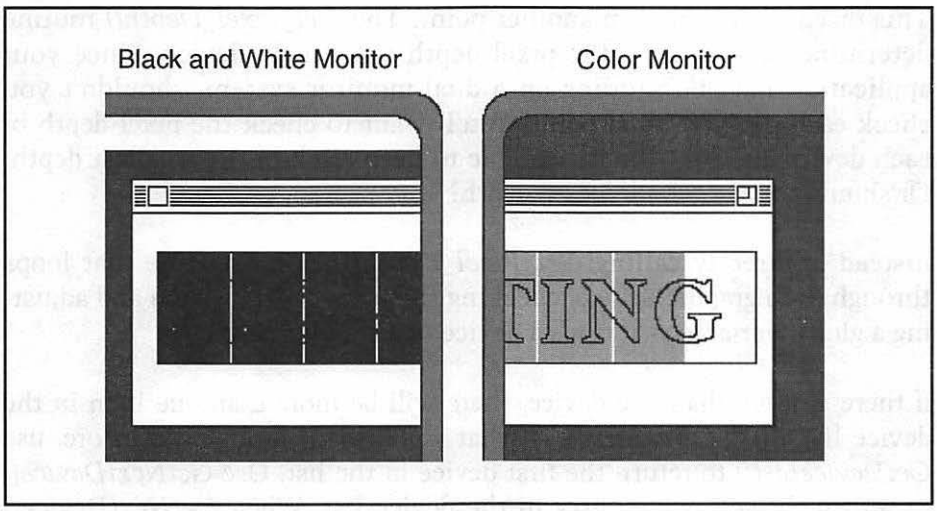


Figure 8–16. *A color picture displayed across two monitors*

To avoid the problem shown in Figure 8–16, you'd display the picture designed for the monitor with the lower pixel depth, 'PICT' 302, as shown in Figure 8–17.

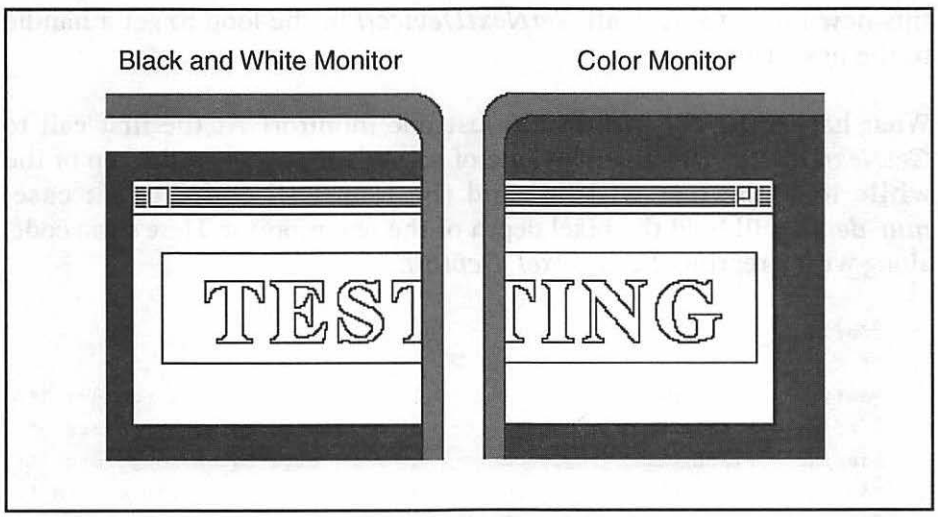


Figure 8–17: *A monochrome picture displayed across two monitors*

The same discussion holds true for a system that has two color monitors; your program should execute as if it were running on a system of the smaller depth.

This discussion brings up another point. The *Get_Pixel_Depth()* routine determines and returns the pixel depth of a single device. Since your application may be running on a dual-monitor system, shouldn't you check each device? Good point. You'll want to check the pixel depth of each device and set a global variable to keep track of the smallest depth. Or should we say "shallowest" depth?

Instead of directly calling *Get_Pixel_Depth()*, call a routine that loops through each graphic device, checking the pixel depth of each and adjusting a global variable as it finds a device of a smaller depth.

If there is more than one device, there will be more than one item in the device list. So the device list is what you'll loop through. As before, use *GetDeviceList()* to return the first device in the list. Use *GetNextDevice()* to return the following entry in the device list. When *GetNextDevice()* returns a value of nil, you know you've reached the end of the list.

Begin by setting the local variable *min_depth* to the largest value you could encounter—24-bit color. Then get a handle to the first device in the device list. Enter a loop and check the pixel depth. If the pixel depth is smaller than the previous low value, reset the *min_depth* to this new lower value. Call *GetNextDevice()* in the loop to get a handle to the next device.

What happens if the system has just one monitor? At the first call to *GetNextDevice()* you'll get a value of nil. When you go to the top of the while loop the test will fail and the loop will end. In that case, *min_depth* will hold the pixel depth of the one monitor. Here's the code, along with a reprint of *Get_Pixel_Depth()*:

```
#define   NIL                      0L

#define   PIXEL_DEPTH_BW            1  /* 1 bit  holds 2  colors  */
#define   PIXEL_DEPTH_4_COLOR       2  /* 2 bits holds 4  colors  */
#define   PIXEL_DEPTH_16_COLOR      4  /* 4 bits holds 16 colors  */
#define   PIXEL_DEPTH_256_COLOR     8  /* 8 bits holds 256 colors */
#define   PIXEL_DEPTH_LOTS_COLOR   16  /* 16 bits holds...uuhmm.. */
#define   PIXEL_DEPTH_MAX_COLOR    24  /* lots and lots of colors */

short  Get_Min_Pixel_Depth( void )
{
   GDHandle  current_device;
```

```
    short     pixel_depth;
    short     min_depth;

    min_depth = PIXEL_DEPTH_MAX_COLOR;
    current_device = GetDeviceList();
    while ( current_device != NIL )
    {
       pixel_depth = Get_Pixel_Depth( current_device );
       if ( pixel_depth < min_depth )
          min_depth = pixel_depth;
       current_device = GetNextDevice( current_device );
    }
    return  min_depth;
}


short  Get_Pixel_Depth( GDHandle  the_device )
{
    PixMapHandle  screenPMapH;
    short         pixel_depth;

    screenPMapH = ( **the_device ).gdPMap;
    pixel_depth = ( **screenPMapH ).pixelSize;
    return  pixel_depth ;
}
```

Note that I define several constants, one for each of the possible values *min_depth* might have. Though they aren't used here, you might use them later on in various tests. Here's an example from a program that runs best on a machine displaying 256 colors. It calls the *Get_Min_Pixel_Depth()* routine to set a global variable, *Min_Pixel_Depth*, to the lowest pixel depth.

```
short  Min_Pixel_Depth;

Min_Pixel_Depth = Get_Min_Pixel_Depth();

switch ( Min_Pixel_Depth )
{
    case PIXEL_DEPTH_BW:
       DrawString( "\pThis program looks better in color!" );
       break;
    case PIXEL_DEPTH_4_COLOR:
    case PIXEL_DEPTH_16_COLOR:
       DrawString( "\pSet your monitor(s) to 256 color if available.");
```

```
        break;
    case PIXEL_DEPTH_256_COLOR:
        DrawString( "\pLeave monitor settings as they are now." );
        break;
    default:
        DrawString( "\pThis program displays only 256 colors." );
        break;
}
```

When to call the pixel depth routines

Near the start of your program you'll want to make a one-time check to see if the system has color QuickDraw. If it does, you'll set a global flag that can be examined by your program at any time. I showed you how to do this earlier when I introduced the *Gestalt()* function.

```
Boolean  Color_QD_Present;
OSErr    err;
long     response;

err = Gestalt( gestaltQuickdrawVersion, &response );

if ( err == noErr )
{
    if ( response == gestaltOriginalQD )
        Color_QD_Present = FALSE;
    else
        Color_QD_Present = TRUE;
}
else
    DrawString("\pGestalt error.");
```

If color QuickDraw is present, determine the lowest color level setting of the attached monitors. If color QuickDraw isn't present you can, or course, safely assume a pixel depth of 1 bit—black and white.

```
if ( Color_QD_Present == TRUE )
    Min_Pixel_Depth = Get_Min_Pixel_Depth();
else
    Min_Pixel_Depth = PIXEL_DEPTH_BW;
```

The Macintosh is a computer with flexible features; there are Macintosh models designed to please all types of users. One of the features that can be varied is the level of colors the monitor will display. As a convenience

to the user, the Monitor's control panel lets the user change the color level at any time, even during the running of your application. Your program should be aware of this and not assume that the level of color at the onset of execution will be the color level throughout the program's entire execution. The Monitor's control panel appears in Figure 8–18.

If the color level can be changed at any time, how can you possibly know when to check to see if the user has made a change? If the user selects the Monitor's control panel while your program is running, an update event will occur. Your program, ever watchful for the occurrence of an event, will be aware of this update event. When an update event occurs, its your cue to check if the color level changed.

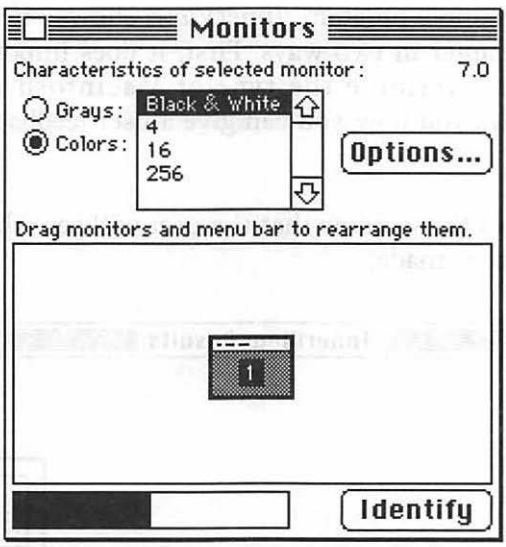


Figure 8–18. *The monitor's control panel*

Place a call to *Get_Min_Pixel_Depth()* within the *updateEvt* case of the *switch* statement in your program's event loop, if color QuickDraw is present. If color QuickDraw isn't present, you know that the value of your global variable *Min_Pixel_Depth*, set to black and white near program startup, won't change. Here's the affected section of the event loop.

```
switch (The_Event.what)
{
```

```
        case mouseDown:
            Handle_Mouse_Down();
            break;

        [ other event types here ]

        case updateEvt:
            if (Color_QD_Present == TRUE)
                Min_Pixel_Depth = Get_Min_Pixel_Depth();
            Handle_Update_Event();
            break;
    }
```

Chapter Program: *InnerView*

This chapter's example program, *InnerView*, shows off the concepts presented in this chapter in two ways. First, it does important behind-the-scenes work to determine the type of Macintosh it's running on. Secondly, it shows you how you can give a user feedback about his own machine.

Figure 8–19 shows the window that the user will see when the program's New menu choice is made.

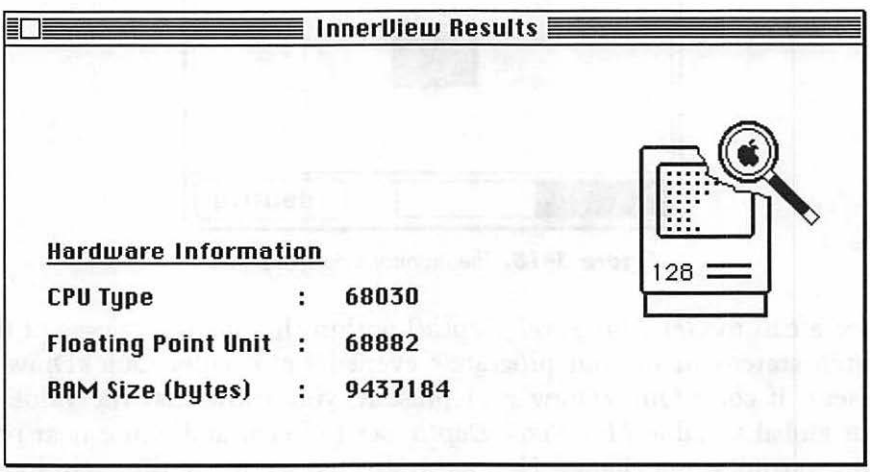


Figure 8–19. InnerView *program in action*

If after looking at Figure 8–19 you feel that I could have made better use of the window's real estate, you're right. But I'm setting things up for Chapter 9. There I expand the capabilities of *InnerView*.

Program resources: *InnerView.π.rsrc*

The *InnerView* program has one 'ALRT' and one 'DITL' resource for displaying the alert that appears when the "About..." menu item choice is made.

InnerView has two 'MENU' resources—one for the menu and one for a File menu. The File menu allows the user to bring up a new *InnerView* results window or quit the program. The two 'MENU' resources are bound together by an 'MBAR' resource.

InnerView has one 'WIND' resource to be used for a window that displays system information. The window also displays a picture. *InnerView* has two 'PICT' resources, though only one will be shown. If the user has a black and white monitor *InnerView* will display 'PICT' 128. If the user has color, the program will put up 'PICT' 129. Figure 8–20 shows the two 'PICT's. 'PICT' 129 will appear in color when viewed in ResEdit. This is the best I can do in a black-and-white book!

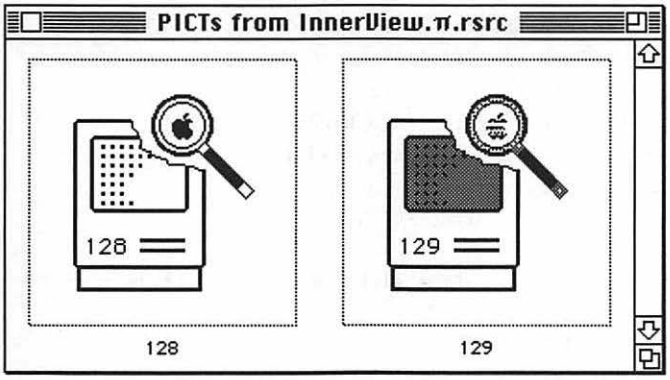


Figure 8–20. *InnerView's two 'PICT' resources*

Program listing: *InnerView.c*

Here, in its entirety, is the *InnerView* source code. Much of it will look familiar to you. The portions not familiar will be covered in the sections that follow.

```
/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include <Traps.h>
#include <GestaltEqu.h>


/*++++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

void   Initialize_Toolbox( void );
void   Check_System( void );
short  Get_Min_Pixel_Depth( void );
short  Get_Pixel_Depth( GDHandle );
void   Set_Window_Drag_Boundaries( void );
void   Set_Screen_Center( void );
void   Set_Up_Menu_Bar( void );
void   Open_InnerView_Window(void);
void   Handle_One_Event( void );
void   Handle_Mouse_Down( void );
void   Handle_Menu_Choice( long );
void   Handle_Apple_Choice( short );
void   Handle_File_Choice( short );
void   Handle_Update( void );
void   Draw_Mac_Picture( void );
void   Draw_Hardware_System_Info_Headings( void );
void   Get_Hardware_Information( void );
void   Close_Window( void );


/*++++++++++++++++++   Define global constants   +++++++++++++++++*/

#define    IV_WIND_ID                128
#define    WIND_WIDTH                460
#define    WIND_HEIGHT               230
#define    STR_LIST_ID               128
#define    WIND_TITLE_STR              1

#define    ABOUT_ALRT_ID             128
#define    TOO_OLD_ALRT_ID           129
```

```
#define    MENU_BAR_ID                128
#define    APPLE_MENU_ID              128
#define       ABOUT_ITEM                1
#define    FILE_MENU_ID               129
#define       NEW_ITEM                  1
#define       QUIT_ITEM                 2
#define    MAC_PICT_BW_ID             128
#define    MAC_PICT_COLOR_ID          129

#define    NIL                         0L
#define    IN_FRONT        (WindowPtr)-1L
#define    REMOVE_EVENTS               0
#define    SLEEP_TICKS                 0L
#define    MOUSE_REGION                0L
#define    DRAG_EDGE                  20

#define    MENU_BAR_HEIGHT            18
#define    PIXEL_DEPTH_BW              1
#define    PIXEL_DEPTH_MAX_COLOR      24

#define    NUM_HARDWARE_HEADINGS       3
#define    LINE_HEIGHT                25
#define    COLUMN_X                  180
#define    HEADING_X                  20
#define    COLON_X                   155
#define    INFO_HEAD_Y               120
#define    PICT_L                    340
#define    PICT_T                     40


/*+++++++++++++++++   Define global variables   +++++++++++++++++*/

Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
Rect         Drag_Rect;
Point        Screen_Center;
WindowPtr    IV_Window_Ptr;
Boolean      Color_QD_Present;
short        Min_Pixel_Depth;
```

```
/*++++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void main( void )
{
   Check_System();
   Initialize_Toolbox();
   Set_Up_Menu_Bar();
   Set_Window_Drag_Boundaries();
   Set_Screen_Center();

   while ( All_Done == FALSE )
      Handle_One_Event();
}


/*++++++++++++++++++  Initialize the Toolbox  ++++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*++++++++++++  Check the machine we're running on  ++++++++++*/

void  Check_System( void )
{
   SysEnvRec  mac_info;
   OSErr      err;
   long       response;

   SysEnvirons( curSysEnvVers, &mac_info );

   if ((mac_info.machineType < 0) || (mac_info.systemVersion < 0x0604))
   {
      StopAlert( TOO_OLD_ALRT_ID, NIL );
      ExitToShell();
   }
```

```
   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap ) );

   err = Gestalt( gestaltQuickdrawVersion, &response );

   if ( err == noErr )
   {
      if ( response == gestaltOriginalQD )
         Color_QD_Present = FALSE;
      else
         Color_QD_Present = TRUE;
   }
   else
      ExitToShell();

   Min_Pixel_Depth = Get_Min_Pixel_Depth();
}


/*+++++++++  Find pixel depth of lowest color monitor  +++++++++*/

short  Get_Min_Pixel_Depth( void )
{
   GDHandle  current_device;
   short     pixel_depth;
   short     min_depth;

   min_depth = PIXEL_DEPTH_MAX_COLOR;
   current_device = GetDeviceList();
   while ( current_device != NIL )
   {
      pixel_depth = Get_Pixel_Depth( current_device );
      if ( pixel_depth < min_depth )
         min_depth = pixel_depth;
      current_device = GetNextDevice( current_device );
   }
   return  min_depth;
}


/*+++++++++++++  Get pixel depth of one monitor  +++++++++++++*/

short  Get_Pixel_Depth( GDHandle the_device )
{
```

```
   PixMapHandle  screenPMapH;
   short         pixel_depth;

   screenPMapH = ( **the_device ).gdPMap;
   pixel_depth = ( **screenPMapH ).pixelSize;
   return  pixel_depth ;
}


/*++++++++++++  Initialize window drag boundaries  ++++++++++++++*/

void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = ( **( GrayRgn ) ).rgnBBox;
   Drag_Rect.left   += DRAG_EDGE;
   Drag_Rect.right  -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}


/*+++++  Determine center of monitor that has the menu bar  ++++*/

void  Set_Screen_Center( void )
{
   Screen_Center.h = screenBits.bounds.right/2;
   Screen_Center.v = (screenBits.bounds.bottom/2) +
((MENU_BAR_HEIGHT/2)/2);
}


/*++++++++++++++++++  Initialize the menu bar  +++++++++++++++++*/

void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      ExitToShell();

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   Apple_Menu = GetMHandle( APPLE_MENU_ID );
   File_Menu  = GetMHandle( FILE_MENU_ID );
```

```
   AddResMenu( Apple_Menu, 'DRVR' );

   DrawMenuBar();
}


/*++++++++++++++++++  Open one results window  ++++++++++++++++++*/

void  Open_InnerView_Window( void )
{
   short  left, top;
   Str255  the_str;

   if ( Color_QD_Present && Min_Pixel_Depth > PIXEL_DEPTH_BW )
      IV_Window_Ptr = GetNewCWindow(IV_WIND_ID, NIL, IN_FRONT);
   else
      IV_Window_Ptr = GetNewWindow( IV_WIND_ID, NIL, IN_FRONT );

   if ( IV_Window_Ptr == NIL )
      ExitToShell();

   GetIndString( the_str, STR_LIST_ID, WIND_TITLE_STR );
   SetWTitle( IV_Window_Ptr, the_str );

   left = Screen_Center.h - ( WIND_WIDTH /2 );
   top  = Screen_Center.v - ( WIND_HEIGHT/2 );
   MoveWindow( IV_Window_Ptr, left, top, TRUE );
   ShowWindow( IV_Window_Ptr );
}


/*++++++++++++++++++  Handle a single event  ++++++++++++++++++*/

void  Handle_One_Event( void )
{
   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
```

```
    {
        case mouseDown:
            Handle_Mouse_Down();
            break;

        case updateEvt:
            if ( Color_QD_Present == TRUE )
                Min_Pixel_Depth = Get_Min_Pixel_Depth();
            Handle_Update();
            break;
    }
}


/*++++++++++++  Handle a click of the mouse button  ++++++++++++++*/

void  Handle_Mouse_Down( void )
{
    WindowPtr  the_window;
    short      the_part;
    long       menu_choice;

    the_part = FindWindow( The_Event.where, &the_window );

    switch ( the_part )
    {
        case inMenuBar:
            menu_choice = MenuSelect( The_Event.where );
            Handle_Menu_Choice( menu_choice );
            break;

        case inSysWindow:
            SystemClick( &The_Event, the_window);
            break;

        case inDrag:
            DragWindow( the_window, The_Event.where, &Drag_Rect );
            break;

        case inGoAway:
            if ( TrackGoAway( the_window, The_Event.where ) )
                Close_Window();
            break;

        case inContent:
```

```
            SelectWindow( the_window );
            break;
      }
}


/*+++++++++++++++++  Handle a click on a menu  ++++++++++++++++++*/

void  Handle_Menu_Choice( long menu_choice )
{
   int  the_menu;
   int  the_menu_item;

   if ( menu_choice != 0 )
   {
      the_menu = HiWord( menu_choice );
      the_menu_item = LoWord( menu_choice );

      switch ( the_menu )
      {
         case APPLE_MENU_ID :
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID :
            Handle_File_Choice( the_menu_item );
            break;
      }
      HiliteMenu( 0 );
   }
}


/*++++++++++++  Handle a click of in the Apple menu  ++++++++++++*/

void  Handle_Apple_Choice( short the_item )
{
   Str255  desk_acc_name;
   int     desk_acc_number;

   switch ( the_item )
   {
      case ABOUT_ITEM :
         NoteAlert( ABOUT_ALRT_ID, NIL );
         break;
```

```
      default :
         GetItem( Apple_Menu, the_item, desk_acc_name );
         desk_acc_number = OpenDeskAcc( desk_acc_name );
         break;
   }
}


/*+++++++++++++  Handle a click in the File menu  +++++++++++++++*/

void  Handle_File_Choice( short the_item )
{
   switch ( the_item )
   {
      case NEW_ITEM :
         if ( IV_Window_Ptr != NIL )
            Close_Window();
         Open_InnerView_Window();
         break;

      case QUIT_ITEM :
         All_Done = TRUE;
         break;
   }
}


/*++++++++++++++++++++  Handle an update event  +++++++++++++++++*/

void  Handle_Update( void )
{
   GrafPtr    old_port;
   WindowPtr  the_window;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   TextFont(0);
   TextSize(12);

   BeginUpdate( IV_Window_Ptr );
      Draw_Mac_Picture();
      Draw_Hardware_System_Info_Headings();
      Get_Hardware_Information();
```

```
   EndUpdate( IV_Window_Ptr );

   SetPort( old_port );
}


/*++++++++++++++  Draw a PICT to the results window  ++++++++++++*/

void  Draw_Mac_Picture( void )
{
   PicHandle  the_pict;
   Rect       pict_rect;
   short      pict_wd, pict_ht;
   GrafPtr    old_port;
   short      pict_id;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   if ( Min_Pixel_Depth > PIXEL_DEPTH_BW )
      pict_id = MAC_PICT_COLOR_ID;
   else
      pict_id = MAC_PICT_BW_ID;

   the_pict = GetPicture( pict_id );

   pict_rect = ( **( the_pict ) ).picFrame;
   pict_wd  = pict_rect.right - pict_rect.left;
   pict_ht = pict_rect.bottom - pict_rect.top;
   SetRect(&pict_rect, PICT_L, PICT_T, PICT_L + pict_wd, PICT_T + pict_ht);

   DrawPicture( the_pict, &pict_rect );

   SetPort( old_port );
}


/*++++++++++++++  Draw headings to the results window  ++++++++++++*/

void  Draw_Hardware_System_Info_Headings( void )
{
   short  colon_y;
   short  i;
   Point  pen_loc;
```

```
    MoveTo( HEADING_X, INFO_HEAD_Y );
    DrawString( "\pHardware Information" );
    GetPen( &pen_loc );
    MoveTo( HEADING_X, INFO_HEAD_Y + 2 );
    LineTo( pen_loc.h, INFO_HEAD_Y + 2 );

    MoveTo( HEADING_X, INFO_HEAD_Y + ( 1 * LINE_HEIGHT ) );
    DrawString( "\pCPU Type");
    MoveTo( HEADING_X, INFO_HEAD_Y + ( 2 * LINE_HEIGHT ) );
    DrawString( "\pFloating Point Unit");
    MoveTo( HEADING_X, INFO_HEAD_Y + ( 3 * LINE_HEIGHT ) );
    DrawString( "\pRAM Size (bytes)");

    colon_y = INFO_HEAD_Y;
    for ( i=1; i <= NUM_HARDWARE_HEADINGS; i++ )
    {
        MoveTo( COLON_X, colon_y + ( i*LINE_HEIGHT ) );
        DrawChar( ':' );
    }
}


/*+++++++++++++++  Get info about user's machine  +++++++++++++++*/

void  Get_Hardware_Information( void )
{
    OSErr     err;
    long      response;
    Str255    byte_str;

    err = Gestalt( gestaltProcessorType, &response );
    if ( err == noErr )
    {
        MoveTo( COLUMN_X, INFO_HEAD_Y + ( 1 * LINE_HEIGHT ) );
        switch ( response )
        {
            case gestalt68000:
                DrawString( "\p68000" );
                break;
            case gestalt68010:
                DrawString( "\p68010" );
                break;
            case gestalt68020:
                DrawString( "\p68020" );
                break;
            case gestalt68030:
                DrawString( "\p68030" );
```

```
            break;
         case gestalt68040:
            DrawString( "\p68040" );
            break;
      }
   }

   err = Gestalt( gestaltFPUType, &response );
   if ( err == noErr )
   {
      MoveTo( COLUMN_X, INFO_HEAD_Y + ( 2 * LINE_HEIGHT ) );
      switch ( response )
      {
         case gestaltNoFPU:
            DrawString( "\pNo FPU present" );
            break;
         case gestalt68881:
            DrawString( "\p68881" );
            break;
         case gestalt68882:
            DrawString( "\p68882" );
            break;
         case gestalt68040FPU:
            DrawString( "\p68040 built-in FPU" );
            break;
      }
   }

   err = Gestalt( gestaltPhysicalRAMSize, &response );
   if ( err == noErr )
   {
      MoveTo( COLUMN_X, INFO_HEAD_Y + (3 * LINE_HEIGHT ) );
      NumToString( response, byte_str );
      DrawString( byte_str );
   }
}


/*++++++++++++++++++   Close one window   ++++++++++++++++++++*/

void  Close_Window( void )
{
   HideWindow( IV_Window_Ptr );
   DisposeWindow( IV_Window_Ptr );
   IV_Window_Ptr = NIL;
}
```

Stepping through the code

Now let's walk through the *InnerView* code, placing emphasis on the new material.

The #include directives

InnerView uses the *Gestalt()* function, so it needs information found in *GestaltEqu.h*. The program uses traps in a call to *NGetTrapAddress()*, so for that it needs Traps.h.

The #define directives

InnerView has a slew of *#defines*. Ready? *IV_WIND_ID* is the resource ID of the program's window. *WIND_WIDTH* and *WIND_HEIGHT* are the window's dimensions, taken from the 'WIND' resource. They'll be used when centering the window. I get the title for the window, *WIND_TITLE_STR*, from a 'STR#' resource with an ID of *STR_LIST_ID*.

ABOUT_ALRT_ID is the 'ALRT' resource for the alert displayed when the "About..." menu item is selected. *TOO_OLD_ALRT_ID* is the 'ALRT' for the alert displayed if the user's machine isn't up to snuff.

InnerView has two menus, so it has two 'MENU' resource. The first is *APPLE_MENU_ID* for the menu, and the second is *FILE_MENU_ID* for the File menu. *ABOUT_ITEM*, *NEW_ITEM*, and *QUIT_ITEM* are the item numbers of the items within the menus. *MENU_BAR_ID* is the resource ID of the 'MBAR'.

The program has two 'PICT' resources; —*MAC_PICT_BW_ID* is a black and white picture, and *MAC_PICT_COLOR_ID* has color. That's it for the resources.

The next several *#defines* are the old standards from previous chapters. *GetNewWindow()* uses *NIL* and *IN_FRONT*. Toolbox initialization makes use of *REMOVE_EVENTS*. *WaitNextEvent()* uses *SLEEP_TICKS* and *MOUSE_REGION*. I use *DRAG_EDGE* to limit window dragging.

The remaining *#defines* are new to *InnerView*. *PIXEL_DEPTH_BW* and *PIXEL_DEPTH_MAX_COLOR* will help determine pixel depth of the monitor.

InnerView draws text in its one window. I use all of the following to help evenly space this text: *NUM_HARWARE_HEADINGS*, *LINE_HEIGHT*, *COLUMN_X*, *HEADING_X*, *COLON_X*, and *INFO_HEAD_Y*. To place the picture just where I want it I use *PICT_L* and *PICT_T*.

```
#define    IV_WIND_ID                  128
#define    WIND_WIDTH                  460
#define    WIND_HEIGHT                 230
#define    STR_LIST_ID                 128
#define    WIND_TITLE_STR                1
#define    ABOUT_ALRT_ID               128
#define    TOO_OLD_ALRT_ID             129

#define    MENU_BAR_ID                 128
#define    APPLE_MENU_ID               128
#define    ABOUT_ITEM                    1
#define    FILE_MENU_ID                129
#define    NEW_ITEM                      1
#define    QUIT_ITEM                     2
#define    MAC_PICT_BW_ID              128
#define    MAC_PICT_COLOR_ID           129

#define    NIL                          0L
#define    IN_FRONT        (WindowPtr)-1L
#define    REMOVE_EVENTS                 0
#define    SLEEP_TICKS                  0L
#define    MOUSE_REGION                 0L
#define    DRAG_EDGE                    20

#define    MENU_BAR_HEIGHT              18
#define    PIXEL_DEPTH_BW                1
#define    PIXEL_DEPTH_MAX_COLOR        24

#define    NUM_HARDWARE_HEADINGS         3
#define    LINE_HEIGHT                  25
#define    COLUMN_X                    180
#define    HEADING_X                    20
#define    COLON_X                     155
#define    INFO_HEAD_Y                 120
```

```
#define     PICT_L                  340
#define     PICT_T                   40
```

The global variables

InnerView uses global variables *All_Done, Multifinder_Present,* and *The_Event* to process events. The program keeps a handle to each of the two menus—*Apple_Menu* and *File_Menu. Drag_Rect* is used for window dragging. The point that is the center of the user's screen is, of course, *Screen_Center. IV_Window_Ptr* will serve to help determine if a window is open. For keeping track of the color setting of the user's monitor I use *Color_QD_Present* and *Min_Pixel_Depth.*

```
Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
Rect         Drag_Rect;
Point        Screen_Center;
WindowPtr    IV_Window_Ptr;
Boolean      Color_QD_Present;
short        Min_Pixel_Depth;
```

The start

InnerView's *main()* function should look familiar; it's much like the *main()* of previous examples. It starts with program initializations and, as always, ends with the event-handling *while* loop.

```
void main( void )
{
   Check_System();
   Initialize_Toolbox();
   Set_Up_Menu_Bar();
   Set_Window_Drag_Boundaries();
   Set_Screen_Center();

   while ( All_Done == FALSE )
      Handle_One_Event();
}
```

As you've seen in past examples, every Macintosh program you write will have some code that is unchanging, such as the *Initialize_Toolbox()* routine called by *main()*.

```
void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}
```

Checking the system

Because the Macintosh scene is constantly changing, you can never be sure on just what kind of Macintosh a program you write will be running. You've got to cover all bases. That's what this chapter is all about. *Check_System()* bundles together into one package much of what is discussed in this chapter. All your programs should have a routine similar to *InnerView*'s *Check_System()* routine and the utility routines it calls. Figure 8–21 summarizes all the things *Check_System()* does.

```
void  Check_System( void )
{
   SysEnvRec  mac_info;
   OSErr      err;
   long       response;

   SysEnvirons( curSysEnvVers, &mac_info );

   if ((mac_info.machineType < 0) || (mac_info.systemVersion < 0x0604))
   {
      StopAlert( TOO_OLD_ALRT_ID, NIL );
      ExitToShell();
   }

   Multifinder_Present = (NGetTrapAddress(_WaitNextEvent, ToolTrap) !=
                          NGetTrapAddress(_Unimplemented, ToolTrap));
```

```
    err = Gestalt( gestaltQuickdrawVersion, &response );

    if ( err == noErr )
    {
       if ( response == gestaltOriginalQD )
          Color_QD_Present = FALSE;
       else
          Color_QD_Present = TRUE;
    }
    else
       ExitToShell();

    Min_Pixel_Depth = Get_Min_Pixel_Depth();
}
```

After *Check_System()* are several functions I call utility routines. These are functions that will appear in many or all of your programs, with little or no modification. *Get_Min_Pixel_Depth()* determines the minimum color level of all monitors connected to the Macintosh. It appears just as it was developed earlier in this chapter. *Get_Min_Pixel_Depth()* calls *Get_Pixel_Depth()*, also covered in this chapter.

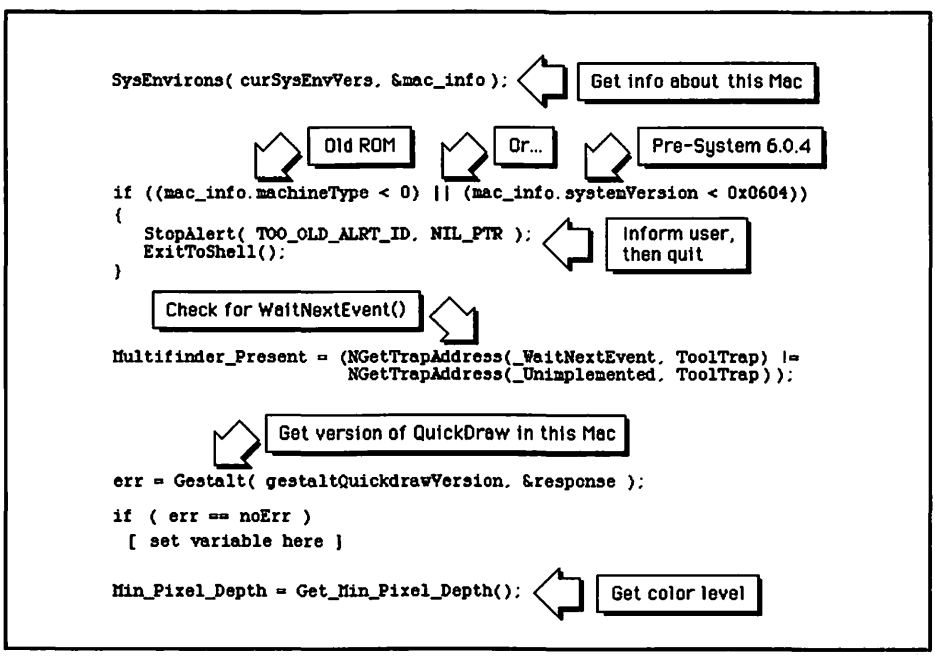


Figure 8-21. *What* Check_System() *does*

```
short  Get_Min_Pixel_Depth( void )
{
   GDHandle  current_device;
   short     pixel_depth;
   short     min_depth;

   min_depth = PIXEL_DEPTH_MAX_COLOR;
   current_device = GetDeviceList();
   while ( current_device != NIL )
   {
      pixel_depth = Get_Pixel_Depth( current_device );
      if ( pixel_depth < min_depth )
         min_depth = pixel_depth;
      current_device = GetNextDevice( current_device );
   }
   return  min_depth;
}


short  Get_Pixel_Depth( GDHandle the_device )
{
   PixMapHandle  screenPMapH;
   short         pixel_depth;

   screenPMapH = ( **the_device ).gdPMap;
   pixel_depth = ( **screenPMapH ).pixelSize;
   return  pixel_depth ;
}
```

Set_Window_Drag_Boundaries() sets the limits that a window can be dragged, based on the desktop area of the Macintosh monitor or monitors.

```
void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = ( **( GrayRgn ) ).rgnBBox;
   Drag_Rect.left   += DRAG_EDGE;
   Drag_Rect.right  -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}
```

This chapter showed two methods of determining the center of the main screen—the screen that holds the menu bar. The first method used a call to *GetMainDevice()* to return a *GDHandle*. Since graphics device routines are part of Color QuickDraw, this method won't work for

monochrome systems. The second method simply uses the QuickDraw global variable *screenBits*. This method works on any system, so that's what I use here.

```
void  Set_Screen_Center( void )
{
   Screen_Center.h = screenBits.bounds.right/2;
   Screen_Center.v = (screenBits.bounds.bottom/2) + ((MENU_BAR_HEIGHT/2)/2);
}
```

Putting up the menu

Set_Up_Menu_Bar() puts the menu bar on the screen. This routine was first covered in Chapter 7.

```
void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      ExitToShell();

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   Apple_Menu = GetMHandle( APPLE_MENU_ID );
   File_Menu  = GetMHandle( FILE_MENU_ID );

   AddResMenu( Apple_Menu, 'DRVR' );

   DrawMenuBar();
}
```

Opening a window

Open_InnerView_Window() will be called a little later on in the program, but I'll cover it here. At startup the program checked to see if the Mac it's running on has color QuickDraw. The program has been con-

stantly on the lookout for a change in the pixel depth of any monitor connected to the system. Here's one reason.

Open_InnerView_Window() checks to see if the system has color QuickDraw and if color is on. If it is the program will display a color picture in the window later on. So you'll want to make a call to *GetNewCWindow()* to get a color window. You don't want to make a call to a color routine like this if color QuickDraw isn't available or if the monitor isn't set to display color.

After opening the window *Open_InnerView_Window()* goes on to set the window's title, as described in Chapter 5. It then centers the window using the center point, *Screen_Center*. Recall that this *Point* variable was calculated at startup in the *Set_Screen_Center()* routine.

```
void  Open_InnerView_Window( void )
{
   short   left, top;
   Str255  the_str;

   if ( Color_QD_Present && Min_Pixel_Depth > PIXEL_DEPTH_BW )
      IV_Window_Ptr = GetNewCWindow(IV_WIND_ID, NIL, IN_FRONT);
   else
      IV_Window_Ptr = GetNewWindow(IV_WIND_ID, NIL, IN_FRONT);

   if ( IV_Window_Ptr == NIL )
      ExitToShell();

   GetIndString( the_str, STR_LIST_ID, WIND_TITLE_STR );
   SetWTitle( IV_Window_Ptr, the_str );

   left = Screen_Center.h - ( WIND_WIDTH /2 );
   top  = Screen_Center.v - ( WIND_HEIGHT/2 );
   MoveWindow( IV_Window_Ptr, left, top, TRUE );
   ShowWindow( IV_Window_Ptr );
}
```

Event handling

InnerView looks for two event types: *updateEvt* and *mouseDown*. This version of *Handle_One_Event()* is similar to previous versions, with one new addition. As mentioned earlier in this chapter, I use the occurrence

of an update event as a signal to check for the pixel depth. Why? If the user selects the Monitor's control panel and changes the color setting of his monitor, it will trigger an update event. How do you know that a particular update event was caused by selecting Monitors and not by some other situation? You don't, so you run the pixel depth check every update, just in case.

```
void  Handle_One_Event( void )
{
   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   switch ( The_Event.what )
   {
      case mouseDown:
         Handle_Mouse_Down();
         break;

      case updateEvt:
         if ( Color_QD_Present == TRUE )
            Min_Pixel_Depth = Get_Min_Pixel_Depth();
         Handle_Update();
         break;
   }
}
```

A click of the mouse is handled by *Handle_Mouse_Down()*. If the click is in the menu bar the *Handle_Menu_Choice()* routine will be called. Nothing new here; everything you see has been covered in Chapters 5 and 7.

```
void  Handle_Mouse_Down( void )
{
   WindowPtr  the_window;
   short      the_part;
   long       menu_choice;

   the_part = FindWindow( The_Event.where, &the_window );
```

```
    switch ( the_part )
    {
       case inMenuBar:
          menu_choice = MenuSelect( The_Event.where );
          Handle_Menu_Choice( menu_choice );
          break;

       case inSysWindow:
          SystemClick( &The_Event, the_window);
          break;

       case inDrag:
          DragWindow( the_window, The_Event.where, &Drag_Rect );
          break;

       case inGoAway:
          if ( TrackGoAway( the_window, The_Event.where ) )
             Close_Window();
          break;

       case inContent:
          SelectWindow( the_window );
          break;
    }
}


void  Handle_Menu_Choice( long menu_choice )
{
    int  the_menu;
    int  the_menu_item;

    if ( menu_choice != 0 )
    {
       the_menu = HiWord( menu_choice );
       the_menu_item = LoWord( menu_choice );

       switch ( the_menu )
       {
          case APPLE_MENU_ID :
             Handle_Apple_Choice( the_menu_item );
             break;

          case FILE_MENU_ID :
```

```
            Handle_File_Choice( the_menu_item );
            break;
        }

        HiliteMenu( 0 );
    }
}
```

A click of the mouse in the menu triggers *Handle_Mouse_Down()* to call *Handle_Apple_Choice()*. This routine is identical to the version created in Chapter 7.

```
void  Handle_Apple_Choice( short the_item )
{
    Str255  desk_acc_name;
    int     desk_acc_number;

    switch ( the_item )
    {
        case ABOUT_ITEM :
            NoteAlert( ABOUT_ALRT_ID, NIL );
            break;

        default :
            GetItem( Apple_Menu, the_item, desk_acc_name );
            desk_acc_number = OpenDeskAcc( desk_acc_name );
            break;
    }
}
```

A mouse click in the File menu tells *Handle_Mouse_Down()* to call *Handle_File_Choice()*. *InnerView* only allows one window on the screen at a time, so if a window is open the program closes it. It then opens the results window with a call to *Open_Inner_View_Window()*.

```
void  Handle_File_Choice( short the_item )
{
    switch ( the_item )
    {
        case NEW_ITEM :
            if ( IV_Window_Ptr != NIL )
                Close_Window();
            Open_InnerView_Window();
```

```
            break;

         case QUIT_ITEM :
            All_Done = TRUE;
            break;
      }
   }
```

After a mouse down event, the second event type handled is an update event. *InnerView* only allows one window on the screen at any given time, so you don't have to do any fancy checks to see what window needs updating. You set *IV_Window_Ptr* to point to the newly opened window when you called *GetNewWindow()* earlier, so that's the window you're working with now.

After setting the port, make a couple of calls to set the text to the system font in a 12 point size. The program nests three routines between calls to *BeginUpdate()* and *EndUpdate()*. I'll cover these three routines next.

```
void  Handle_Update( void )
{
   GrafPtr    old_port;
   WindowPtr  the_window;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   TextFont( systemFont );
   TextSize(12);

   BeginUpdate( IV_Window_Ptr );
      Draw_Mac_Picture();
      Draw_Hardware_System_Info_Headings();
      Get_Hardware_Information();
   EndUpdate( IV_Window_Ptr );

   SetPort( old_port );
}
```

With only one window, why bother setting the port? Because there are other ports on the screen, including the screen itself! Always keep track of ports as is done here.

The screen, or desktop, is a port. If you don't set the port there's a good chance that any drawing you do will end up on the desktop, not in your window. You can see for yourself by commenting out the *SetPort(IV_Window_Ptr)* line in *InnerView.c* and recompiling the program.

Draw_Mac_Picture() draws a picture in the *InnerView* window. If color QuickDraw is present and color is turned on, the program displays the color 'PICT' stored in the resource file. If you're running in monochrome, the program displays a different 'PICT'—one drawn to show up better in black and white.

If you have a color computer, try running *InnerView* both with the color on and off to verify that a different picture appears. How can you be sure that the color 'PICT' isn't being shown (in a monochrome format) when your computer is in the black-and-white mode? Note that I included the 'PICT' ID right on the picture itself. In monochrome you should see 'PICT' 128, in color mode you'll see 'PICT' 129. Refer back to Figure 8–20 to see the two 'PICT's. Of course 'PICT' 129 will appear in color when viewed in ResEdit or on a color computer.

The rest of *Draw_Mac_Picture()* is the same as seen in Chapter 3.

```
void  Draw_Mac_Picture( void )
{
   PicHandle  the_pict;
   Rect       pict_rect;
   short      pict_wd, pict_ht;
   GrafPtr    old_port;
   short      pict_id;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   if ( Min_Pixel_Depth > PIXEL_DEPTH_BW )
      pict_id = MAC_PICT_COLOR_ID;
   else
      pict_id = MAC_PICT_BW_ID;
```

```
    the_pict = GetPicture( pict_id );

    pict_rect = ( **( the_pict ) ).picFrame;
    pict_wd  = pict_rect.right - pict_rect.left;
    pict_ht = pict_rect.bottom - pict_rect.top;
    SetRect(&pict_rect, PICT_L, PICT_T, PICT_L + pict_wd, PICT_T +
pict_ht);

    DrawPicture( the_pict, &pict_rect );

    SetPort( old_port );
}
```

The *Draw_Hardware_System_Info_Headings()* routine is nothing more than a series of pen movements and text drawing calls to display some unchanging text in the window.

```
void  Draw_Hardware_System_Info_Headings( void )
{
    short  colon_y;
    short  i;
    Point  pen_loc;

    MoveTo( HEADING_X, INFO_HEAD_Y );
    DrawString( "\pHardware Information" );
    GetPen( &pen_loc );
    MoveTo( HEADING_X, INFO_HEAD_Y + 2 );
    LineTo( pen_loc.h, INFO_HEAD_Y + 2 );

    MoveTo( HEADING_X, INFO_HEAD_Y + ( 1 * LINE_HEIGHT ) );
    DrawString( "\pCPU Type");
    MoveTo( HEADING_X, INFO_HEAD_Y + ( 2 * LINE_HEIGHT ) );
    DrawString( "\pFloating Point Unit");
    MoveTo( HEADING_X, INFO_HEAD_Y + ( 3 * LINE_HEIGHT ) );
    DrawString( "\pRAM Size (bytes)");

    colon_y = INFO_HEAD_Y;
    for ( i=1; i <= NUM_HARDWARE_HEADINGS; i++ )
    {
        MoveTo( COLON_X, colon_y + ( i*LINE_HEIGHT ) );
        DrawChar( ':' );
    }
}
```

Now, the meat of the program. *Get_Hardware_Information()* depends on *Gestalt()* to get a few of the features of the machine *InnerView* is running on. *InnerView* checks for the processor type, the floating-point unit type, and the amount of RAM in the computer. Once you understand how one check is made, you can easily add more of your own. In fact, that's exactly what you'll do in next chapter's example program.

```
void  Get_Hardware_Information( void )
{
   OSErr    err;
   long     response;
   Str255   byte_str;

   err = Gestalt( gestaltProcessorType, &response );
   if ( err == noErr )
   {
      MoveTo( COLUMN_X, INFO_HEAD_Y + ( 1 * LINE_HEIGHT ) );
      switch ( response )
      {
         case gestalt68000:
            DrawString( "\p68000" );
            break;
         case gestalt68010:
            DrawString( "\p68010" );
            break;
         case gestalt68020:
            DrawString( "\p68020" );
            break;
         case gestalt68030:
            DrawString( "\p68030" );
            break;
         case gestalt68040:
            DrawString( "\p68040" );
            break;
      }
   }

   err = Gestalt( gestaltFPUType, &response );
   if ( err == noErr )
   {
      MoveTo( COLUMN_X, INFO_HEAD_Y + ( 2 * LINE_HEIGHT ) );
      switch ( response )
```

```
        {
            case gestaltNoFPU:
                DrawString( "\pNo FPU present" );
                break;
            case gestalt68881:
                DrawString( "\p68881" );
                break;
            case gestalt68882:
                DrawString( "\p68882" );
                break;
            case gestalt68040FPU:
                DrawString( "\p68040 built-in FPU" );
                break;
        }
    }

    err = Gestalt( gestaltPhysicalRAMSize, &response );
    if ( err == noErr )
    {
        MoveTo( COLUMN_X, INFO_HEAD_Y + (3 * LINE_HEIGHT ) );
        NumToString( response, byte_str );
        DrawString( byte_str );
    }
}
```

Closing a window

A mouse click in the window or a "New" menu choice closes the window. Because I left it up to the Window Manager to assign memory storage for the window I use *DisposeWindow()* rather than *CloseWindow()* and *DisposPtr()*. The global *WindowPtr* variable *IV_Window_Ptr* is set to nil so that the program knows that no window is open on the screen.

```
void  Close_Window( void )
{
    HideWindow( IV_Window_Ptr );
    DisposeWindow( IV_Window_Ptr );
    IV_Window_Ptr = NIL;
}
```

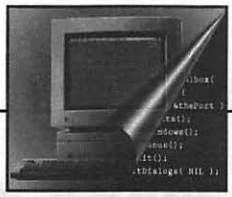
Chapter Summary

The thousands of Toolbox routines exist in the ROM chips of your Macintosh. A Toolbox routine is also called a trap, and each trap has a trap number. When you include a call to a Toolbox routine in your code, the trap number for that routine tells the processor where in memory it will find the code that makes up that routine.

As Macintosh computers are improved, so is the ROM. New versions of ROM contain new Toolbox calls, and thus new trap numbers. Many of the Toolbox functions you call will have been present in the ROM of the first Macintosh computer, and in every Macintosh since. Some routines you'll want to use, however, only reside in more recent versions of ROM. It's up to you to determine if the computer your program is running on supports the calls you're going to make.

The *Gestalt()* routine is your most powerful means of determining the contents of ROM. By passing it the name of a routine, preceded by an underscore, you can see if that routine exists on any given Macintosh. If it doesn't, you'll want to either use a substitute routine or exit the program and return to the Finder.

Gestalt() can also be used to determine many different hardware and software features of the machine your program is running on. By passing *Gestalt()* different selector codes, you can find out whether a Macintosh supports color, what version of the system is installed, the amount of RAM in the computer, and a host of other environmental factors.



9 Memory Management

Chapter 2, Macintosh Memory, gave you the background you need to understand Macintosh memory management techniques. In this chapter you'll delve deeper into the topic. Here I present the code and specific techniques that will help you avoid system crashes.

The code for Macintosh programs must be grouped into segments no larger than 32 K each. If a program gets larger than that—and it will—segmentation becomes a concern. In this chapter you'll learn how to properly segment your program code.

Here you'll get insight into selecting the proper amount of memory to allocate for your application's partition. You'll see how to set the partition in THINK C and how to make changes to it using the 'SIZE' resource.

Macintosh Memory Management

The term *memory management* refers to the allocation, movement, tracking, and removing of things in memory. These "things" I speak of are most often resources. You know that menus, dialogs, and windows all start out as resources. Your program's code itself is turned into 'CODE' resources that get loaded and moved in memory. This book refers to these things generically as *objects* in memory.

At the heart of memory management is the Macintosh Memory Manger. The Memory Manager does much of the "behind the scenes" work to keep track of what is going on in RAM. It also provides the programmer with a set of routines to assist in memory management tasks. Because the Macintosh uses memory management techniques not found on most other computers, programmers new to the Macintosh often inject memory-related bugs into their programs. A thorough understanding of how the Macintosh works with memory, as described in Chapter 2, along with the more specific programming techniques described in this chapter, will help you reduce the number of bugs of this type.

Objects in memory can have different *attributes* applied to them. This chapter discusses these attributes in some detail. For now, here's an overview.

A block can be *relocatable.* A relocatable block potentially can be moved about in memory and released from memory by the Memory Manager, without any intervention by your program. The Memory Manager does this when memory is scarce. A block can also be marked *nonrelocatable.* If a block is nonrelocatable it is fixed in memory; the Memory Manager will not ever move it or purge it on its own. It can only be released from memory by your program explicitly calling a Toolbox routine to dispose of it.

If a block is relocatable it can be either *locked* or *unlocked.* A locked block cannot be moved in memory. If it's unlocked, it can be shuffled about in memory during compaction. If it's unlocked, it potentially can also be removed from memory by the Memory Manager. Although today even many low-cost Macs come equipped with 4 Mb of memory, memory remains a scarce resource. Why? The size of applications has grown at an equal pace. So, regardless of the amount of memory on the Mac your completed program is running on, your code is likely to be shuffled around in memory.

If a block is relocatable and unlocked it can be made either *purgeable* or *unpurgeable*. If it's purgeable, the Memory Manager can release it from memory if memory becomes scarce. If the object is important enough to remain in memory even during times of memory shortage, it can be marked as unpurgeable.

Figure 9–1 shows the different attributes that can be imposed on a block. Notice that if a block is marked as nonrelocatable it can't be unlocked or purged.

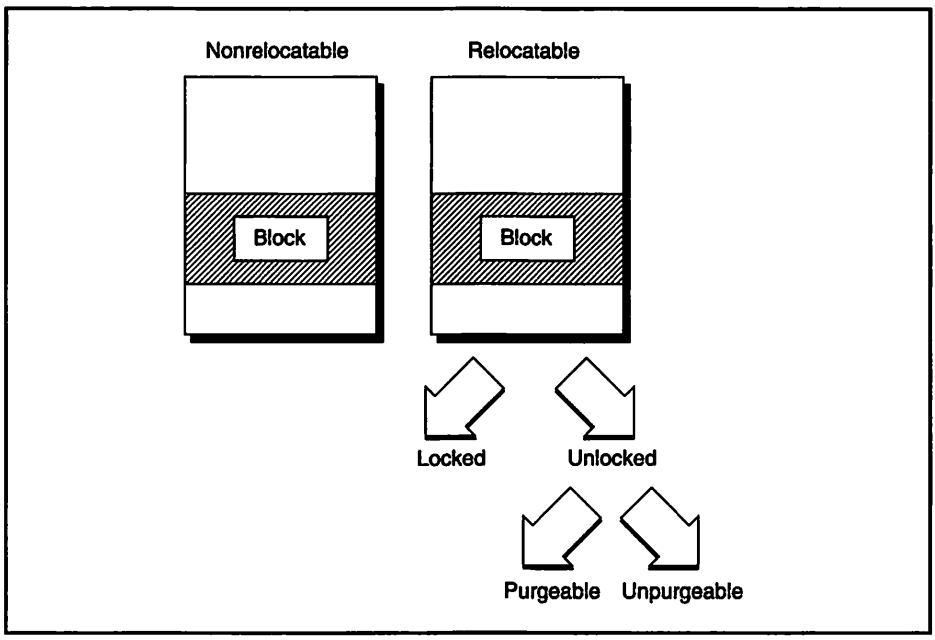


Figure 9–1. *Attributes of a Block in Memory*

Avoiding Heap Fragmentation

Chapter 2 discussed how your application's heap can become fragmented as your program runs. Objects are loaded into memory and then stay where they are, get moved, or are purged—removed. The objects that don't move (the nonrelocatable objects) can play havoc on your program's execution. They cause roadblocks in the heap that prevent efficient use of memory. This heap fragmentation can literally kill a program. In this section you'll learn how fragmentation can be minimized.

How nonrelocatable blocks are created

A block in memory can be marked as relocatable or nonrelocatable. The Memory Manager can move blocks that are relocatable in the heap. Blocks that are nonrelocatable always stay in one place, even when memory is being compacted.

You have only a limited amount of control over allocating nonrelocatable blocks. Any call you directly make to *NewPtr()* creates one. Additionally, your program will indirectly call *NewPtr()* when it calls some Toolbox routines. *GetNewWindow()* is such a call. *GetNewWindow()* loads a window in memory. A call to *GetNewWindow()* also makes a call to *NewPtr()* to create the *WindowPtr* that it returns to your program. The *WindowPtr* points to the nonrelocatable block that holds a *WindowRecord*. Figure 9–2 shows this.

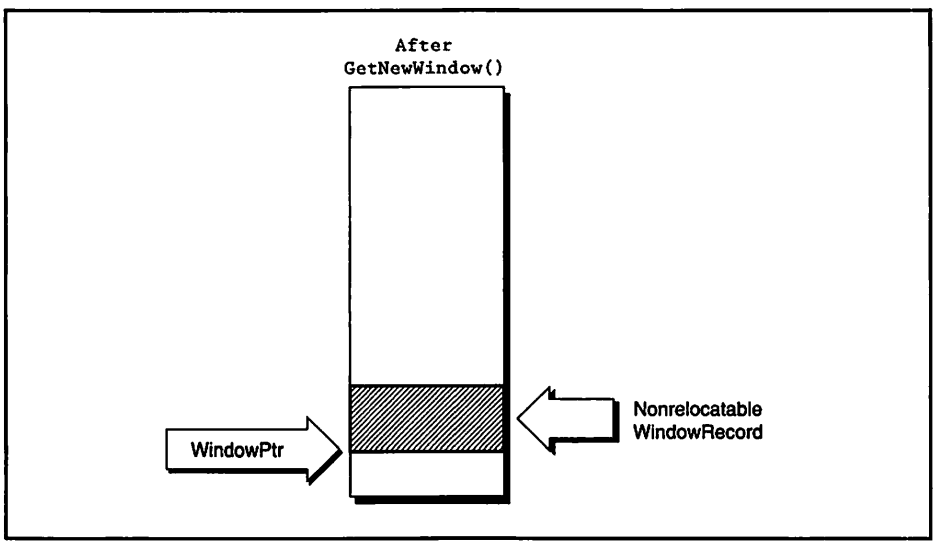


Figure 9–2. *The window structure is nonrelocatable*

The Memory Manager will attempt to place a newly-created nonrelocatable block as low as possible in the heap. But if it is placed above relocatable blocks, and those blocks are eventually purged, an island is formed. A nonrelocatable block—no matter how small it is—creates an obstruction in memory. Figure 9–3 illustrates this. And because the block is nonrelocatable, heap compaction won't help the situation.

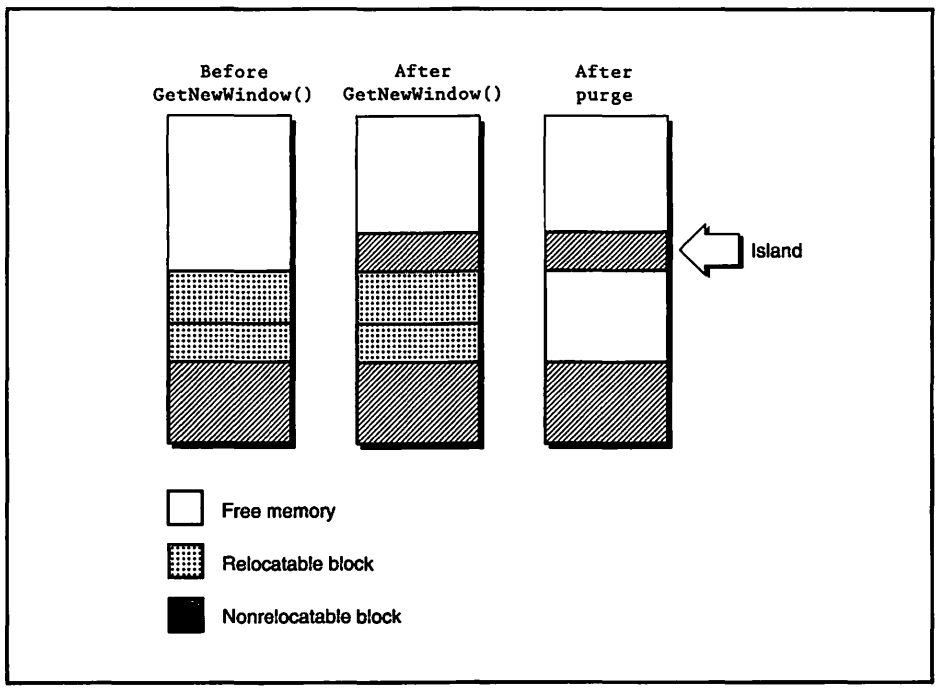


Figure 9–3. *A* WindowRecord *creates an island in memory*

When a window is closed with a call to *DisposeWindow()*, the nonrelocatable block is removed from memory. That's good. But that could be too late. While the window is open, an attempt to load a large object into memory could fail. And some programs will keep one or more windows open for the entire duration of the program, making what could be a partial solution entirely obsolete.

It should be obvious to you by now that you'd like to avoid nonrelocatable blocks. However, you don't want to go to such lengths as minimizing the number of windows in your programs; windows are what the Macintosh is all about. Fortunately there is a way out of this dilemma, and I'll discuss that next.

Reserving memory to reduce fragmentation

The next best thing to avoiding a nonrelocatable block is participating in its placement. If you can control where the block goes, you can place it

as low as possible in memory. That way it won't be an obstruction later on as the Memory Manager attempts to load other objects into memory.

When your program first starts up, some of its program code is loaded into the application partition. (I have more to say about this when I talk about segments later in this chapter.) After the code is loaded you can immediately reserve storage for your window even though it hasn't been opened yet. If you reserve this memory very early in your program's execution, the memory will be low in memory; that's the desirable position for it. Later, when *GetNewWindow()* is called, you'll specify the storage to use and you'll specify the reserved block of memory. This is shown in Figure 9–4.

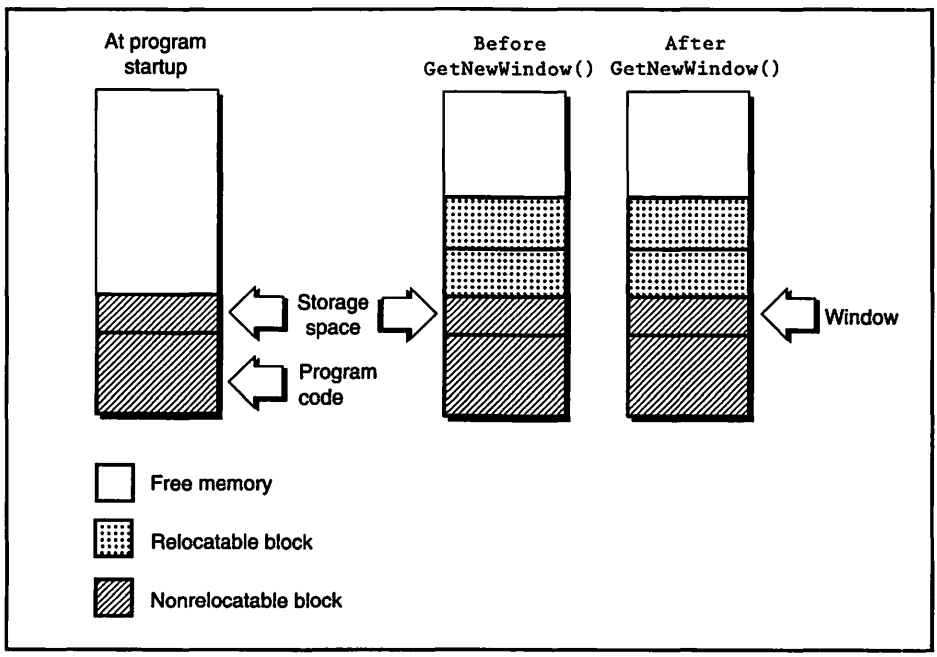


Figure 9–4. *Reserving a block of memory before it's needed*

What if the window you've reserved storage for isn't opened until much later on in your program? Doesn't this storage space then go wasted until that time? True enough. But you aren't trying to save on memory here. You're trying to avoid fragmentation. Your memory storage may be as few as a hundred bytes or so. If you created a window without using storage, the resulting fragmentation brought on by a one-hundred-byte window could make thousands and thousands of bytes unusable.

Thousands and thousands of bytes? Sure. It depends on what your program is attempting to load. Imagine your application is up and running. It has 40 K of free space, divided into two 20 K areas by an island. If your program tries to load a 30 K 'PICT' resource, it will fail. The program won't crash, but the picture won't be displayed.

In case you're wondering, the above situation of a 30 K picture is not at all unreasonable. Especially if your program has color pictures in its resource file.

How do you go about reserving memory? You already know because you did it back in Chapter 5. Yes, I really was leading up to something! Here's what you did back then:

```
#define     WIND_ID              128
#define     IN_FRONT    (WindowPtr)-1

WindowPtr   new_window;
Ptr         wind_storage;
short       left, top;

wind_storage = NewPtr( sizeof ( MyWindRecord ) );
new_window = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
```

Recall that *MyWindRecord* was your modified window record structure. You can do the same thing for a standard *WindowRecord*:

```
wind_storage = NewPtr( sizeof ( WindowRecord ) );
new_window = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
```

NewPtr() reserves a nonrelocatable block of memory. The size of the block is the size of whatever you specify in *sizeof()*. Chapter 5 introduced you to window storage. Now, it's time to make it really useful. Instead of calling *NewPtr()* just before you open your window, call it immediately after your program starts up. The result will be just as shown back in Figure 9–4. Save the resulting *Ptr* in a global variable. Then, when it's time to open a window, pass *GetNewWindow()* this pointer.

Let's say you're writing a program that will open a window to which the user can draw. Optionally, the user can open a second window that will display a graph of some data he has entered. With the possibility of two windows being opened, you know that you should reserve space for two *WindowRecords* right off the bat. Here's a code fragment that reserves memory for two windows.

```
Ptr  Draw_Wind_Storage;
Ptr  Graph_Wind_Storage;

void  main( void )
{
   Initialize_Toolbox();
   Reserve_Window_Memory();

   while ( All_Done == FALSE )
      Handle_One_Event();
}

void Reserve_Window_Memory( void )
{
   Draw_Wind_Storage  = NewPtr( sizeof ( WindowRecord ) );
   Graph_Wind_Storage = NewPtr( sizeof ( WindowRecord ) );
}

void  Open_Draw_Window( void )
{
   WindowPtr  new_window;

   new_window = GetNewWindow( WIND_ID, Draw_Wind_Storage, IN_FRONT );

   [ more code here ]
}

void  Open_Graph_Window( void )
{
   WindowPtr  new_window;

   new_window = GetNewWindow( WIND_ID, Graph_Wind_Storage, IN_FRONT );

   [ more code here ]
}
```

Speaking of assigning window storage, do you recall the two different ways to close windows?

If you allocate the storage yourself, call *CloseWindow()* and *DisposPtr()*. *CloseWindow()* doesn't release the memory you carefully set aside for the *WindowRecord*. That way if the user chooses the same command that created the window in the first place, you can again use that space. *DisposPtr()* releases the memory occupied by the pointer to the window. You aren't concerned about those four bytes.

If you let the Mac set the storage for you by passing a nil value for the storage, then call *DisposeWindow()*. *DisposeWindow()* frees up all memory associated with the window. You didn't set it aside, so you shouldn't care that it's freed.

```
#define    NIL    0L

WindowPtr  use_my_mem_window;
WindowPtr  use_mac_mem_window;
Ptr        wind_storage;

/* using my own storage */
My_Wind_Storage = NewPtr( sizeof ( WindowRecord ) );
use_my_mem_window  = GetNewWindow( WIND_ID, My_Wind_Storage, IN_FRONT );
CloseWindow( use_my_mem_window );
DisposPtr( ( Ptr ) use_my_mem_window );

/* using Mac-supplied storage */
use_mac_mem_window = GetNewWindow( WIND_ID, NIL, IN_FRONT );
DisposeWindow( mac_store_window );
```

Lesson 9-1: Reserving Memory

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Reserving Memory

Setting aside window storage early in the execution of your program is a way of reserving memory for nonrelocatable blocks. You can also reserve a small amount of memory that will help your program as it works with relocatable blocks.

Allocating master pointer blocks

From Chapter 2 you know that a master pointer is a special pointer. Like any pointer, it points to an object. But unlike a normal pointer, a master pointer can track moving objects in memory, not just fixed objects. A *WindowPtr* is an example of a normal pointer. It points to a fixed, nonrelocatable *WindowRecord.* A master pointer, on the other hand, points to relocatable blocks.

How is a relocatable block formed? Through a call to *NewHandle().* When your program calls *NewHandle()* it returns a handle. A handle contains the address of a master pointer. Figure 9–5, repeated from Chapter 2, summarizes what a handle is.

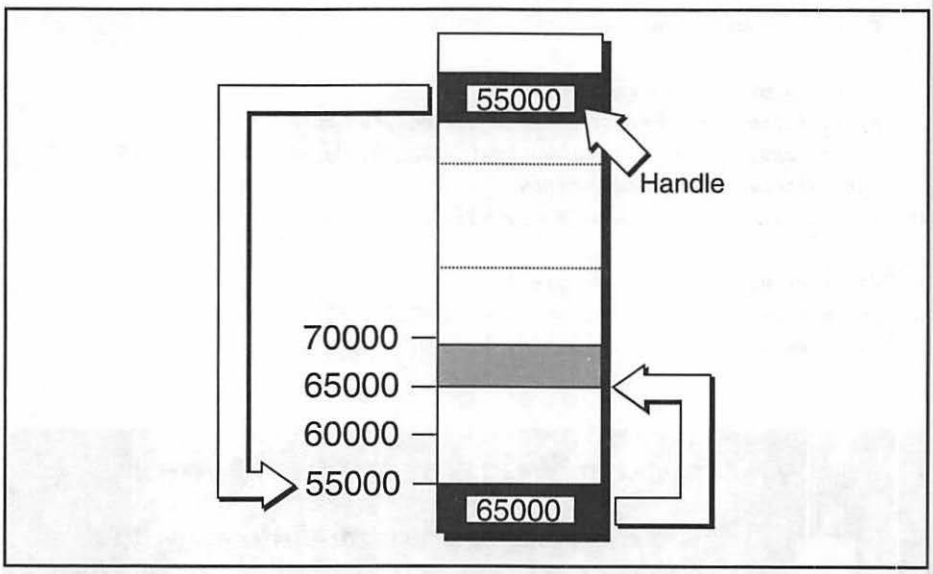


Figure 9–5. *A handle holds the address of a master pointer*

Now that you know that the master pointer itself is fixed in memory, you should also realize it would be advantageous to have it fixed low in memory. That would help reduce fragmentation. Just as you reserved memory to hold nonrelocatable *WindowRecord* blocks, you'll want to reserve memory to hold master pointers.

The Macintosh uses *master pointer blocks* to hold master pointers. A master pointer block is a contiguous area set aside for 64 master pointers. When your program starts up, the Memory Manager creates one master pointer block for your program's use. It does this immediately so that this nonrelocatable block is placed low in memory. You can create an additional master pointer block by doing what the Memory Manager does: call the Toolbox routine *MoreMasters()*. Figure 9–6 illustrates this.

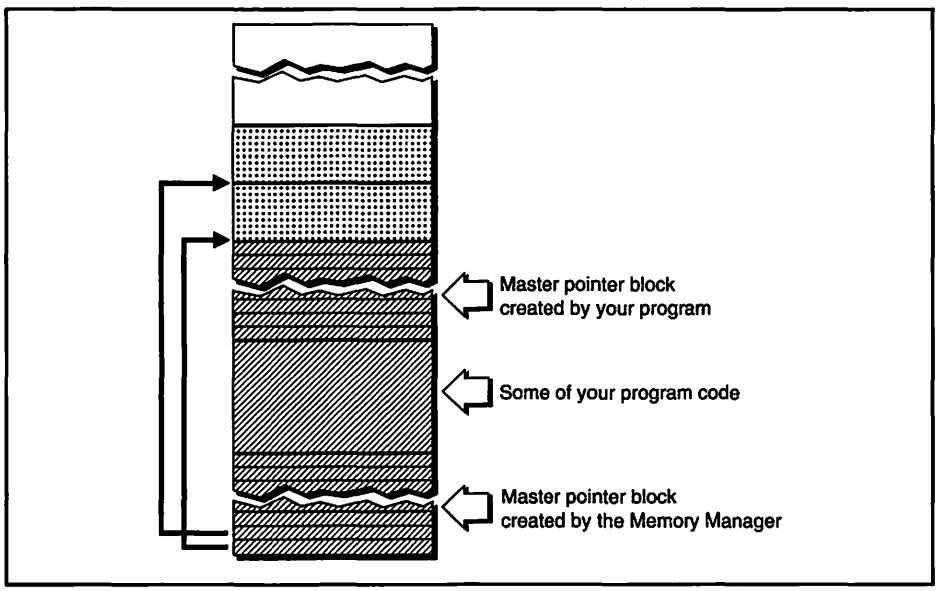


Figure 9–6. *Master pointer blocks in memory*

At the bottom of memory in Figure 9–6 is the master pointer block created by the Memory Manager. Some of the code from your program is above that. Which part of your code that gets loaded immediately is a topic of discussion later in this chapter. A second master pointer block is above the code. This block was created by your program. The code that is loaded in memory made a call to *MoreMasters()* to create this second master pointer block. The figure also shows relocatable blocks in memory. Each has a master pointer pointing to it.

One master pointer points to one relocatable block of memory. One master pointer block is thus capable of pointing to 64 relocatable blocks. It may seem unlikely that your program would call *NewHandle()* more than 64 times, so the issue of how many times you should call *MoreMasters()* might seem unimportant. But there's more to the story then I've told you so far.

When you reserve memory for *WindowRecords*, you do so based on the number of windows your program will open. To reserve memory for master pointers you should base the number of master pointers on the number of relocatable blocks that your program will use; that is, blocks created by calls to *NewHandle()*. How do you do this? It's not as easy as counting the number of times you use *NewHandle()* in your source code; you might not even call it. But the Toolbox will. The Toolbox calls *NewPtr()* in response to a call to *GetNewWindow()*. By the same token the Toolbox calls *NewHandle()* in response to several Toolbox calls. Some Toolbox calls result in two or three calls to *NewHandle()*. All this makes calculating the number of calls to *NewHandle()* difficult.

In determining how many times to call *MoreMasters()* you should keep the following thoughts in mind. A pointer always holds an address, and an address on the Macintosh always occupies four bytes. Thus a pointer is always four bytes in size, regardless the size of the block it points to. That means a single master pointer block, which holds 64 master pointers and an eight byte header, is always 264 bytes in size.

From the preceding paragraph you know that a master pointer block does not occupy a lot of memory. You also need to know that a nonrelocatable object, no matter how small, can cause fragmentation. Whenever possible you want to allocate nonrelocatable objects low in memory where they can do the least amount of damage.

From these two ideas you may accurately draw the conclusion that it is better to call *MoreMasters()* too many times than too few. Programmers generally call *MoreMasters()* about four times. Including the block that the Memory Manager creates, that gives a program five master pointer blocks.

You want your master pointer blocks low in memory, so you want to make the calls to *MoreMasters()* right away. Like this:

```
main()
{
```

```
    MoreMasters();
    MoreMasters();
    MoreMasters();
    MoreMasters();

    Initialize_Toolbox();

    [ and so forth... ]
}
```

Setting the heap size

When your application first starts up, its application heap is set to a small size. As your program requires more memory the Memory Manager will gradually increase the size of the heap. This method of heap expansion can lead to fragmentation. A much more efficient method of enlarging the application's heap is to enlarge everything all at once, at program startup. Conveniently, there's a Toolbox routine that does just that. *MaxApplZone()* should be one of the first calls your program makes. By expanding the heap all at once, you know that future memory allocations will be carried out much more quickly. Here's how your *main()* routine should look, now that you know about *MaxApplZone()* and the *MoreMasters()* routine covered in the previous section:

```
main()
{
    MaxApplZone();

    MoreMasters();
    MoreMasters();
    MoreMasters();
    MoreMasters();

    Initialize_Toolbox();

    [ and so forth... ]
}
```

If *MaxApplZone()* is so important, why didn't I use it earlier in some of the example programs? Because all of the example programs have been small, and memory management hasn't been much of an issue. And because I don't like to introduce new topics until I reach the appropriate chapter!

Writing 32-bit Clean Programs

The number of bits used to hold an address determines how many addresses can be accessed. Before System 7, 24-bit addressing was used. That allowed the Mac to access a maximum of 8 Mb of RAM. With the arrival of System 7 came 32-bit addressing. Using 32 bits to hold an address allows accessing up to 1 GigaByte of RAM.

In previous versions of system software, only 24 of the 32 bits of a pointer or handle were used to hold a memory address. The remaining eight bits were either ignored or used to store additional information. The bits in a master pointer are an example.

A master pointer holds the starting address of a block in memory. The highest bit of a master pointer keeps track of whether the block is locked in memory. The lower 24 bits of the master pointer give the starting address of the block. Figure 9–7 illustrates this. Remember that lower addresses are at the bottom of RAM; that's why the block's starting address is pictured at the bottom of the block.

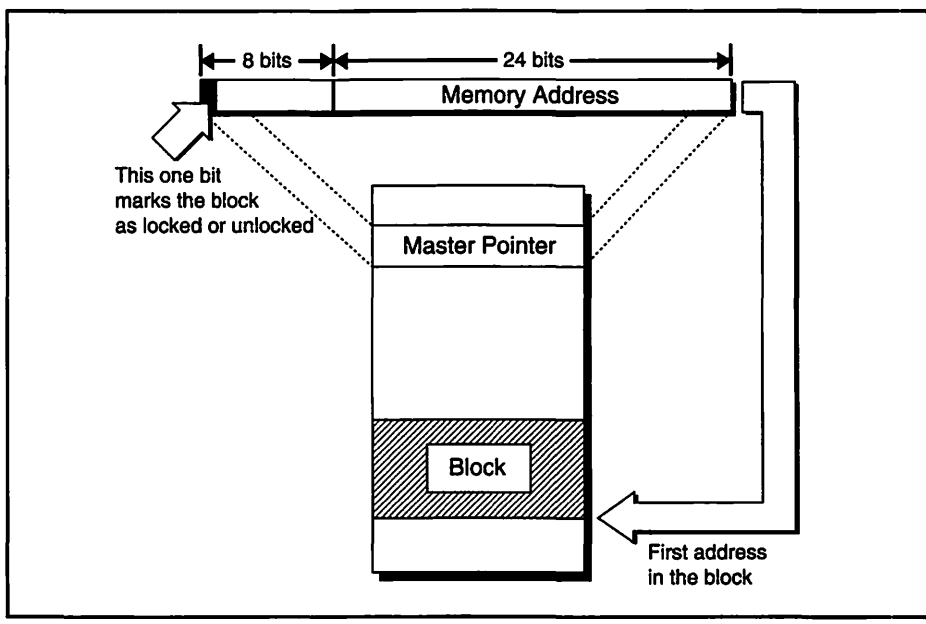


Figure 9–7. *Bits of a master pointer*

With the advent of System 7, using the upper eight bits of a pointer for anything but part of an address is now discouraged. When the Memory Manager looks at 32 bits, it assumes that all 32 bits comprise an address. Other information stored in some of these bits will not be recognized by the Memory Manager; that information is assumed to be part of an address. The results, of course, can be disastrous.

Programs that are written with no extraneous information in any of the 32 bits of an address are said to be 32-bit clean. That is, they will run cleanly on a Macintosh that is using 32-bit addressing.

To allow you to run programs that aren't 32-bit clean, the Memory Control Panel lets you switch between 24-bit and 32-bit addressing in System 7. It can do this because ROMs that contain a 32-bit Memory Manager also contain, for compatibility reasons, a 24-bit Memory Manager. The downside is that with your Mac set to 24-bit addressing, only 8 Mb of RAM will be accessible, even if you have more than that.

Because of the migration towards System 7, you'll want all of your programs to be 32-bit clean. Bits in master pointers used for purposes other than addressing are the primary cause for an application not to be 32-bit clean. This was an acceptable practice for pre-System 7 programs, but not anymore.

Don't become alarmed. If you don't try anything real tricky, your programs will most likely be 32-bit clean. Take the example pictured in Figure 9–7. If you use the Toolbox routine *HLock()* to lock a block, you're fine. *HLock()* won't do what's shown in Figure 9–7—change a bit in the master pointer. It instead stores this information elsewhere.

If you don't use the Toolbox routine *HLock()* and instead use your knowledge of what the bits in a master pointer look like (or used to look like) then try to set or clear the upper bit using direct bit manipulation, your program will no longer be considered 32-bit clean.

Master pointer bit manipulation is one source of breaking 32-bit clean standards. Another is using customized window definition functions and customized control definition functions—resources of type 'WDEF' and 'CDEF'. Definition functions let you create your own types of windows and controls that differ from the standard types. Both of these topics are beyond the scope of this book. If you plan to use either custom window or custom control definitions, make sure your reference sources were written with System 7 and 32-bit clean addressing in mind.

How can you be sure your program is in fact 32-bit clean? Test it. Thoroughly test your program on a Macintosh that has System 7. Check the Memory Control Panel and make sure that 32-bit addressing is turned on. If it isn't, turn it on and reboot the system. Then run your program, testing each aspect of it.

"Testing each aspect of it" is something you'd want to do with or without the issue of 32-bit addressing, right?

Segmentation

We've said several times in this book that just about everything in a Mac program ends up being a resource. You create several of these resources yourself when you add a 'WIND', 'DITL', 'DLOG', or any other resource to your program's resource file.

The compiler you use also creates resources when it builds an application from your source code. The compiler turns all of your compiled source code into 'CODE' resources and stores them in the application. Remember *VeryBasics*, the example program from Chapter 1? Figure 9–8 shows the resources for the *VeryBasics* application.

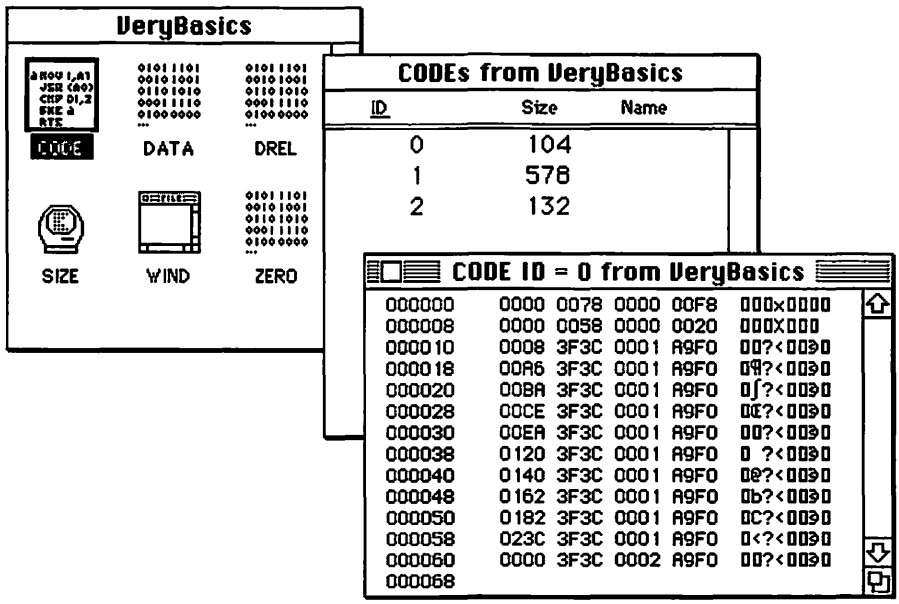


Figure 9–8. *'CODE' resources in the VeryBasics application*

Look familiar? It shouldn't! You remember what the resource file for *VeryBasics* looked like, not the resources in the application itself. The *VeryBasics.π.rsrc* file contained just a single 'WIND' resource. It's shown in Figure 9–9.

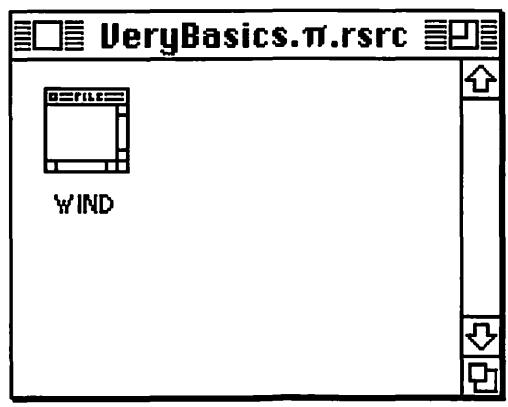


Figure 9–9. *The* VeryBasics.π.rsrc *file for VeryBasics*

Some further explanation is in order. You might want to refer to Figure 9–10 during this discussion. I borrowed it from Chapter 1. When you

build your final application, the compiler merges the compiled source code with the resources in your resource file. The result is a standalone application. When the link is complete, the only thing the user needs to run your program is the program itself. The user doesn't need your source code and doesn't need the resource file. Everything has been incorporated into the application file.

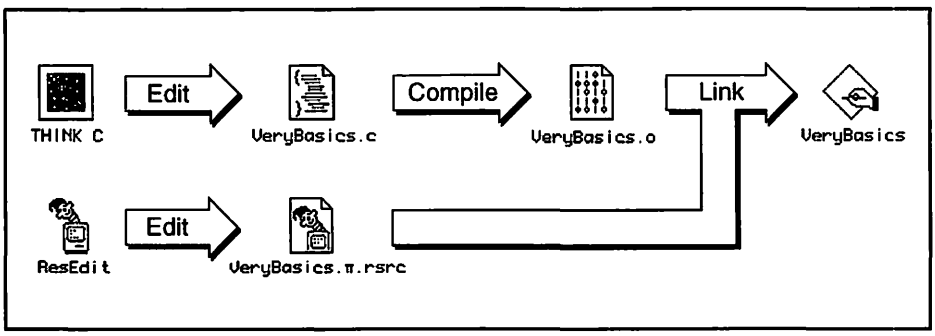


Figure 9–10. *The program creation cycle*

You've used ResEdit to add and edit resources in a resource file. You can also use ResEdit to view the resource composition of a stand-alone program—any program. Let's run ResEdit and open the *VeryBasics* program, not the *VeryBasics.π.rsrc* resource file. The contents are shown in Figure 9–11.

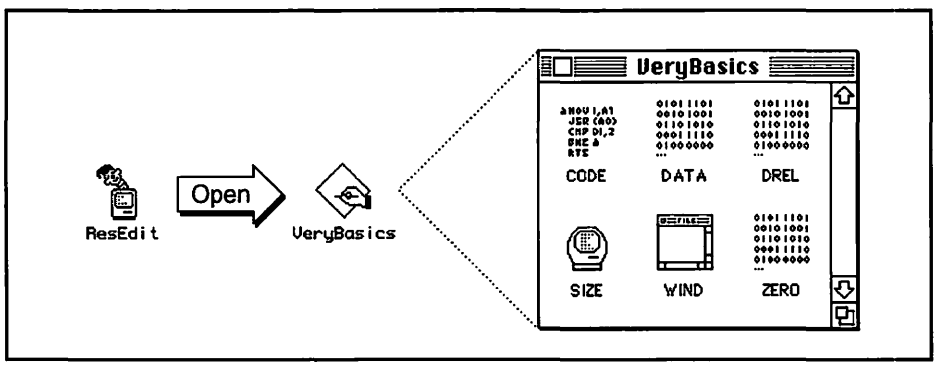


Figure 9–11. *Looking at the resources of the* VeryBasics *application*

Very interesting. The 'WIND' resource is there, all right. But so are several other resources. Your compiler is responsible for adding them. Most important to you is the addition of 'CODE' resources. That's your source

code, compiled and packaged into segments. Finally, the point of all of this: each 'CODE' resource is a segment. *VeryBasics* happens to have three of them, with resource IDs of 0, 1, and 2, shown back in Figure 9–8.

Why have more than one 'CODE' resource—more than one segment? Because of a 32 K size limitation imposed on a segment. One segment can't exceed 32 K. In one way this is bad; in another way it's good. What's bad is you have to be concerned with segments. What's good is that segments permit the existence of huge programs, even programs that are bigger than the memory available on the Macintosh it's running on. How so? The segments of a program aren't all loaded into memory at one time. They get swapped in and out of memory as a program executes so that an entire program doesn't have to reside in memory.

If you've programmed MS Windows you may have optimized memory usage by creating working sets. A working set is a division of a program that performs a task, or related tasks—a segment. Windows terminology for the loading and unloading of these sets is dynamic linking.

Now that you know just what segments are, take a look at how to create them in THINK C. THINK C, like any Macintosh compiler you use, lets you decide how you want to segment, or divide up, your program. After that you'll learn how to decide just what to put in each segment.

Segmenting a program in THINK C

Segments are composed of source files. You can have more than one source file in a segment, but the code in a single file cannot be separated into two segments. Figure 9–12 illustrates this by showing two ways of segmenting a program that has three source code files.

Up to this point all of the example programs have consisted of just one source file. Like any programming environment, Macintosh or not, THINK C lets you divide your source code across multiple files. This Chapter's example program, *InnerViewII*, does just that.

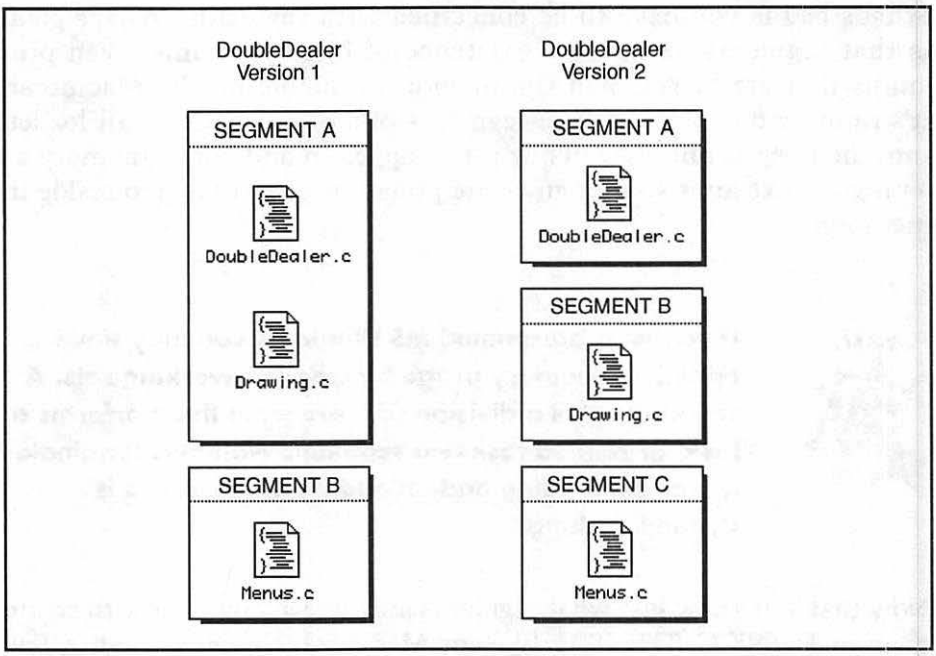


Figure 9–12. *Looking at the resources of the VeryBasics application*

Figure 9–12 shows two possible ways of segmenting the *DoubleDealer* program, a fictitious Macintosh game of duplicity and deceit. Which way is correct? Both. Which way is better? That depends on the routines that are in each source file. The next section looks at a technique for determining what files to group together. This section covers the process of segmenting (a very simple one,) using THINK C.

DOS programmers, you're going to love this. If you've programmed the 8086 you know segments as 64 K chunks of addressable memory space. You also know a lot of mumbo-jumbo about relative offsets, relative addresses, and segmented addresses. Forget it all, now.

Windows programmers, you too will like Macintosh segments. You're used to the SEGMENTS keyword and the module definition file. There's nothing analogous to that for the Mac. You too can forget what you know about the process of segmenting a program.

Figure 9–13 shows the project window for a program called *DoubleDealer*. The program's three source files, and the ever-present *MacTraps*, are all in one segment: Segment 2.

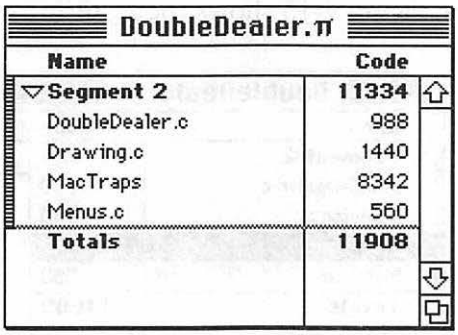


Figure 9–13. *The DoubleDealer project with one Segment*

The fact that THINK C gives the first segment in your program the name Segment 2 might make you think there's a Segment 1 hiding somewhere. There is. Segment 1 is called the *main segment*, and every Macintosh program has one. Program segments get loaded, unloaded, and shuffled around in memory as a program runs. But not Segment 1. It serves as a foundation. It stays in memory from program startup to program termination.

There's also a Segment 0, also present in every program. It's a table that holds information about your program, such as the memory requirements of the global variables in your program.

More than one segment can be in memory at one time. If a program needs to run code from Segment A, the segment is loaded into memory and locked into place. When execution of the code is complete, Segment A isn't immediately removed from memory. Instead, it is unlocked and marked as purgeable. That means that if a different segment, Segment B, needs to be loaded, and there isn't enough available memory, Segment A will be purged from memory to provide room for Segment B.

The default segmentation that THINK C provides is fine for a small program like *DoubleDealer*. For larger programs you'll want to rearrange and regroup your source files into segments of your own choosing. To create a new segment simply click on a file and drag it beneath the Totals line. In Figure 9–14 I clicked on *MacTraps*, and now I'm dragging it to the Totals area. Figure 9–15 shows the resulting segmentation.

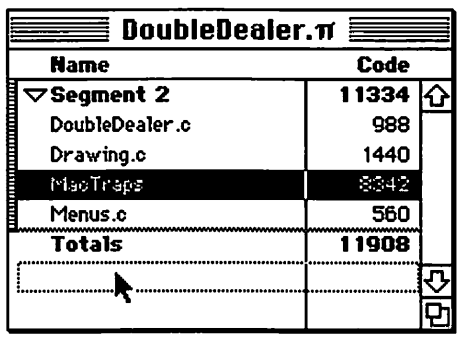


Figure 9–14. *Moving* MacTraps *into a new segment*

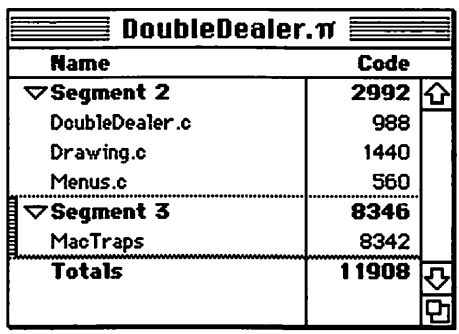


Figure 9–15. *Segmentation after creating a new segment with* MacTraps

To move a file from a segment to an already existing segment, simply click on it and drag it into that segment.

Determining how to segment a program

The overriding reason to segment a program is that you have to! The 32 K addressing barrier demands that your program, if its going to compile into something greater than 32 K, be divided into more than one segment. A second reason to segment is for memory efficiency. If you give a little thought to how you're going to group your program's source code files into segments, you can improve how your program uses memory.

You don't *have* to give any thought to how you segment a program. You could group source files in any way you want, as long as no segment code size exceeds 32 K. If you're used to programming a machine other than a Mac, the temptation to do that may be great. But the Macintosh and THINK C make segmenting so easy, there's every reason to put a little planning into program segmentation.

I've stressed that when a segment is loaded into memory, all of the routines in all of the source files in that segment are loaded into memory. So it makes sense for you to group related routines together, both in source files and in segments.

Let's look at a practical example. Imagine that, due to user-feedback, you've greatly enhanced the graphics capabilities of the *DoubleDealer* program. You've rereleased it as Version 2.0. Available, of course, for the modest upgrade fee of $79.95. *DoubleDealer* now is capable of displaying three different screens of graphics. It might show one, two, or all three of the screens in succession. Figure 9–16 now shows the segmented program.

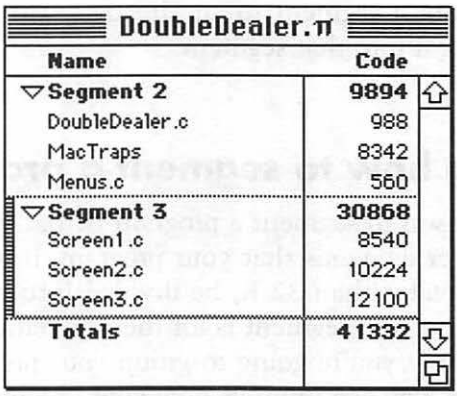


Figure 9–16. *Segmentation of Version 2.0 of DoubleDealer*

To display the first of three screens, *DoubleDealer* will load Segment 3 into memory. If the second screen is to be displayed, the code for it is already in memory because it's in the same segment. What would happen if *Screen1.c* and *Screen2.c* files were in separate segments? After displaying the first screen the program needs to access the code that draws the second screen. If this segment is not already in memory (and it might not be) the program would have to access the hard drive to load the segment. Remember, the entire program might not be in memory. One or more segments may still be on disk.

A disk access is the slowest activity a program will perform. You try to reduce this slowness through thoughtful segmentation.

When you look at Figure 9–16 you can see that the total code for *DoubleDealer* has reached a size of 41,332 bytes. Since this is greater than the 32 K segment limit, segmenting *DoubleDealer* is no longer an option; its a necessity. And since you have to shuffle the files around anyway, why not spend a few minutes planning out a logical grouping arrangement?

To avoid the shuffling of segments being shuffled in and out of memory, you try to keep code that works together grouped together. Sometimes code falls into obvious groupings; sometimes it doesn't.

There are a couple of "rules of thumb" you can use as you make decisions about segmenting your program. The first was just discussed here. To achieve the second requires a very minimal amount of effort on your part; you'll see how in the next section.

- Put routines that are used in conjunction with one another in the same segment. If a menu selection or a click on a button results in the calling of a half dozen routines, place those six routines together in one segment.

- Load code that is constantly used, like the event-loop, into memory right when the program starts. That places it low in memory. Leave this code in memory for the duration of your program's execution.

The main segment

Segment 1, which results in 'CODE' resource 1, is called the *main segment*: every Macintosh program has one. This segment is preloaded into memory when your program starts. That places it in the desirable position near the bottom of the application heap where it won't cause fragmentation, as objects loaded higher in memory might. I said segment 1 is like a foundation. It stays in memory from program startup to program termination. That's because the block of memory that it occupies is locked and unpurgeable. It won't be moved, and won't be unloaded, ever. See Figure 9–17.

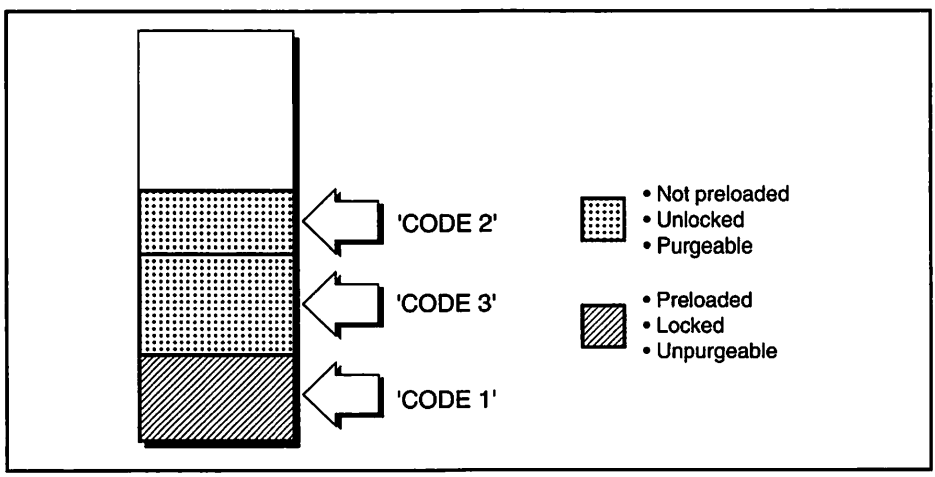


Figure 9–17. *'CODE' resources loaded into the application partition*

Besides the main segment, Figure 9–17 also shows a 'CODE' 2 and 'CODE' 3 resource loaded. Notice that these two resources are "out of order." Code isn't loaded sequentially by resource number. It is loaded as it's used. This might not, and probably won't, correspond to the segment numbers of the code.

Because Segment 1 is unpurgeable, any code within this segment is always in memory; execution of this code doesn't require disk access.

Notice in a THINK C project that there is no Segment 1 listed. That's because the compiler creates 'CODE' 1. It holds a small amount of what is called "foundation" code. Your code starts at Segment 2, which will become 'CODE' resource 2. Notice in Figure 9–17 that 'CODE' 1 is locked into place and unpurgeable, while the two other segments that have been loaded, 'CODE' 2 and 'CODE' 3, aren't. Normally, the segments you divide your THINK C project into are purgeable. But you can control that.

Blocks of memory have attributes; I've been discussing them for quite a while now. THINK C lets you set the attributes of the block of memory to which each segment will be loaded. By double clicking on a segment name in the project window, you open a dialog that lets you set the attributes for that segment. Figure 9–18 shows this.

he default settings for a segment are also shown in Figure 9–18. A segment starts out as protected. That means its contents can't be changed. The segment is marked as locked, which means it won't be moved in memory. And it is marked as purgeable, meaning that the Memory Manager can remove it from memory if it's not in use and memory is getting low.

The edit text box shown in Figure 9–18 allows you to name a segment. If you don't give it a name, you'll see the segment number displayed in the project window. If you do name it, you'll see that name in place of the number. Renaming the segment is optional. The compiler will ignore the segment names; those are only for your benefit.

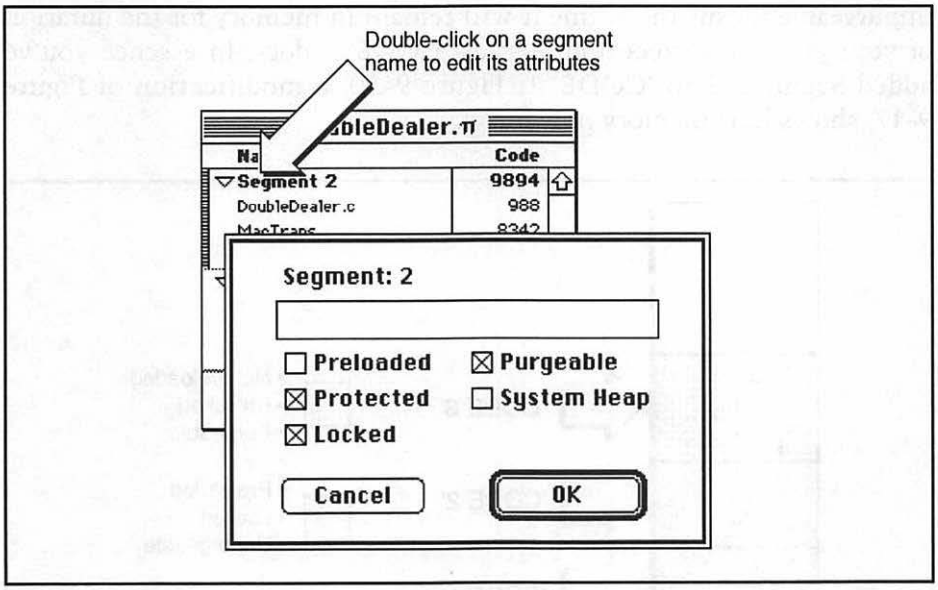


Figure 9–18. *Setting the attributes of a segment in THINK C*

Now it's time to change the attributes of a segment. You'll change the settings of a segment to those shown in Figure 9–19.

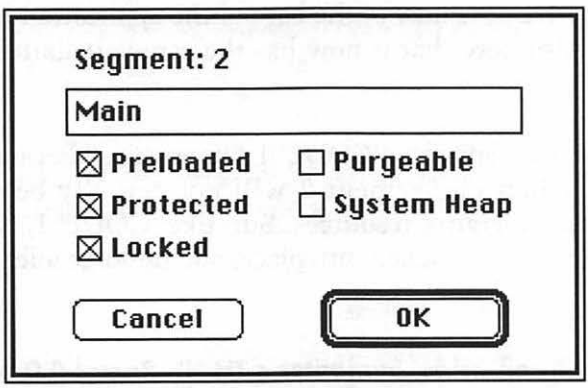


Figure 9–19. *Changing a segment's attributes*

In Figure 9–19 you've changed Segment 2 from purgeable to unpurgeable by unchecking the Purgeable check box. You've also designated the segment to be preloaded. Then you called the segment "Main." What will the results of these changes be? Preloading a segment places it in memory at program start up, just as 'CODE' 1 is. Marking a segment as

unpurgeable means the segment will remain in memory for the duration of your program's execution, just as 'CODE' 1 does. In essence you've added Segment 2 to 'CODE' 1. Figure 9–20, a modification of Figure 9–17, shows how memory now looks.

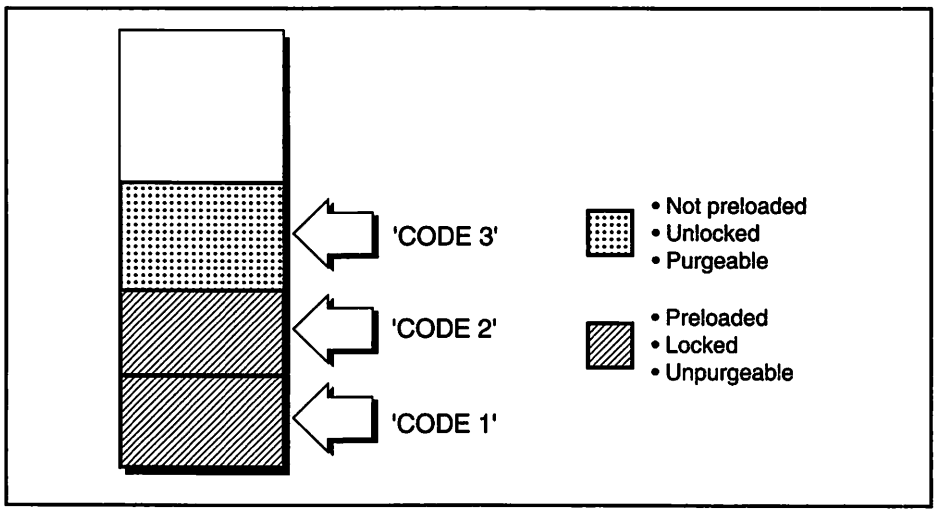


Figure 9–20. *Loaded 'CODE' resources after attribute changes*

Notice that 'CODE' 2 is now at the base of the application heap, adjacent to 'CODE' 1. Also note that it now has the same attributes as the main segment.

Segment 2 has been added to 'CODE' 1 "in essence" because that is the effect you've achieved; Segment 2 will not actually be merged with 'CODE' 1 in the program's resources. But, like 'CODE' 1, 'CODE' 2 will now be low in memory, locked into place, and unpurgeable.

As an aside, Symantec's THINK Pascal 4.0 has a segment called "Main" in each project window. Moving files into that segment in the project window has the effect of literally adding that code to 'CODE' 1. When the project is compiled into an application, the size of the application's 'CODE' 1 resource varies depending on the source files designated to appear in that segment.

By changing the attributes of Segment 2 you have, in effect, expanded the size of the main segment. That's why the segment has the name "Main" in the THINK C project. Now, any source files you put in Segment 2, now called Main, will be preloaded and locked into memory. That's the ideal place for code that's executed often—code that otherwise might frequently be moving in and out of memory. If memory becomes tight during program execution, segments are swapped in and out of memory. That's disk access, and that's what you're trying to avoid. Any code in your newly-created "Main" segment will be immune from this swapping.

So what's the big deal? Why not just make all your segments preloaded, locked, and unpurgeable. Then allow your program a big enough partition for all the code to fit in memory. Like one giant "Main." That way *everything* gets loaded and stays put; no disk accesses to load 'CODE' as your program runs! You're forgetting one thing; you're a thoughtful programmer. The user of your program may want to run other programs at the same time that yours is running. Your program shouldn't hog all the RAM on the user's computer! By allowing segments to be purgeable, your program can occupy a much smaller amount of RAM than it could if everything is loaded together.

In the previous section I said that one of the objectives of planning your program segmentation was to group related routines into the same segment. I said the other objective is to load often-used code low in memory and keep it there. Now you know how to achieve this second objective.

You'll want to put your main event loop in Segment 2. For the examples used here that means putting the *main()* function and the *Handle_One_Event()* function in a source file to be put in Segment 2. Here are the two routines:

```
void  main( void )
{
   Initialize_Toolbox();
   Initialize_Variables();
```

```
    while ( All_Done == FALSE )
        Handle_One_Event();
}

void  Handle_One_Event( void )
{
    [ handle event here ]
}
```

Examine your source code to see which routines are called often. Those routines are candidates for a place in Segment 2. Also look for functions that are called from various parts of your program. Because several routines call a function like this, and because you are limited to the 32 K segment size, you might not be able to place this function and all the functions that call it into one segment. This, too, is a candidate for a place in Segment 2.

You don't want to put code that is seldom called in Segment 2. *Initialize_Toolbox()* is one example of seldom-called code. Your program only calls this routine once. After you call it you don't want it stuck permanently in memory.

This brings us back to the idea of simply preloading everything into memory to avoid moving code in and out of memory. Your program might have a large amount of code that is only executed once during the running of your program. In this case, there is absolutely no need to have all your code in memory. After the code is executed, and if memory becomes tight, the 'CODE' resource that holds this one-time-executable code will be purged to provide room for a different 'CODE' resource. You'll gladly tolerate a disk access in this case. Since the one-time-only code will never be called again, it will never be loaded again. That frees up a lot of memory that would be tied up forever if you had marked the segment as unpurgeable.

I've discussed segment loading and unloading quite a bit. Let's next look at the part you play in the control of this process.

Unloading segments

When your program attempts to call a routine that is not in memory, the segment where the routine is located will be loaded into memory—without any help from you. The Memory Manager handles that. When execution of the code within the segment is complete, you'll want to release, or unload, the segment. The releasing of a segment is not automatically done for you. You'll do that with calls to the Toolbox routine *UnloadSeg()*.

UnloadSeg() doesn't actually remove a segment from memory. It just lets the Mac know that it has permission to purge the segment. You saw earlier that THINK C, by default, marks each segment as locked. That means the Memory Manager can't move it and can't purge it. *UnloadSeg()* changes those attributes so that the Memory Manager knows it can move or release the block if space is needed.

If a program is not large, every segment may fit into the program's memory partition. In that case, segments will be loaded as they're needed, but won't ever be unloaded. In that case memory isn't limited, so there won't be any need to remove segments from memory. That's why you might see fully-functional Macintosh source code written by others without ever seeing a call to *UnloadSeg()*.

Since segments may be moved in, out, and back into memory, how do you determine when to call *UnloadSeg()* for any particular segment? That's easy; you don't have to. If you call *UnloadSeg()* for every segment in your program, you're assured of marking each segment as purgeable at the end of every pass through the main event loop. Calling *UnloadSeg()* at each pass through the main event loops causes the routine to be called an inordinate number of times during the course of your program's execution. Take a look at Figure 9–21, then read why this apparent overkill is necessary.

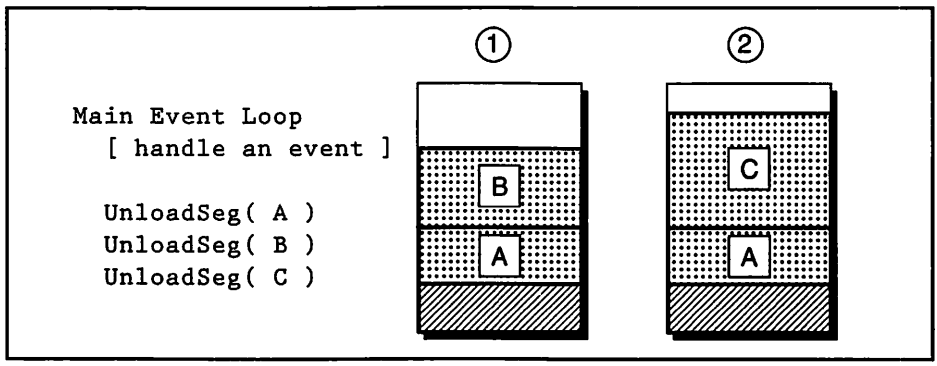


Figure 9–21. UnloadSeg() *and two "snapshots" of memory*

Figure 9–21 shows the application heap at two different points during the run of a hypothetical program. The program consists of three segments, called A, B, and C. At point #1 segments A and B are in memory. The main event loop comes to an end, and *UnloadSeg()* is called on each of the three segments. Though segment C isn't in memory, a call to *UnloadSeg()* on it has no effect. Now segments A and B have been marked as purgeable; the Memory Manager has permission to remove either or both if it needs the memory they occupy.

Next, the program makes a call to a routine in segment C. Segment C, (and the routine that is in it) isn't in memory. Figure 9–21 shows that there is not enough memory in the application's heap to support all three segments. Because the segments are marked as purgeable, segment B can be purged. It is, segment C is loaded, and you're now at point #2 in the figure.

The program reaches the end of another pass through the main event loop. *UnloadSeg()* is again called on each segment. Why, if all the segments were marked as purgeable, is this again necessary? Because, when a segment is loaded it has the attributes of being locked and unpurgeable. Even if it was previously loaded and marked as purgeable by a call to *UnloadSeg()* on it, it comes into memory each time as unpurgeable. At point #1 the calls to *UnloadSeg()* affected segments A and B. At point two they affect segment C.

You won't be able to figure out at every point in your program's execution which segments are in memory, which are about to be loaded, and which are to be unloaded. That's why you call *UnloadSeg()* on every segment in your program, and every pass through the main event loop. No, this won't slow your program down. Remember, a call to *UnloadSeg()* doesn't actually unload the segment, it just toggles a couple of bits that let the Memory Manager know it's all right to unload that segment. That all happens quickly.

Let's look at the latest version of *DoubleDealer*—version 3.0—as an example of how *UnloadSeg()* is called. I've added a long animation sequence to the program, in a source file called *Screen4.c*, and put it into its own segment. Segment 2 has become the Main segment. I pulled all of *DoubleDealer*'s initialization code out of *DoubleDealer*.c and put it in its own file and its own segment. Figure 9–22 shows how the project is shaping up.

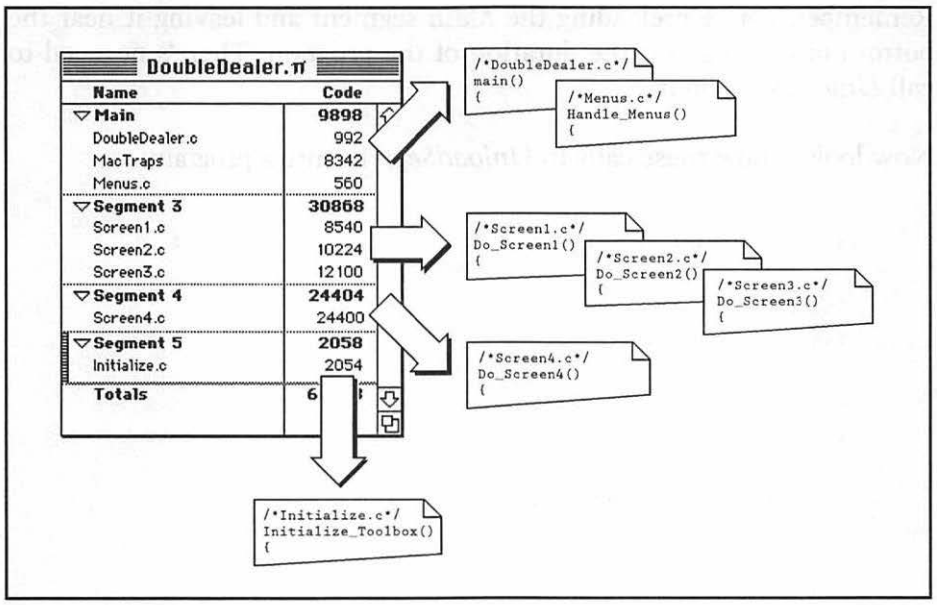


Figure 9–22. *DoubleDealer THINK C project*

Notice in Figure 9–22 that I didn't put the *Initialize.c* file in Segment 4, even though I could have without exceeding the 32 K segment size limit. I put it in its own segment because I know it will only be called once. I don't want this code with *Screen4.c* code, which might be in and out of memory several times.

A call to *UnloadSeg()* requires one parameter: a pointer to a function in the segment to unload. This can be any function in the segment. For *UnloadSeg()*, simply using the function's name as the parameter serves as the pointer to it.

Here's the code that would unload each of the segments in *DoubleDealer*, except for the Main segment:

```
UnloadSeg( Do_Screen1 );
UnloadSeg( Do_Screen2 );
UnloadSeg( Do_Screen3 );
UnloadSeg( Do_Screen4 );
UnloadSeg( Initialize_Toolbox );
```

Remember, you're preloading the *Main* segment and leaving it near the bottom of the heap for the duration of the program. There's no need to call *UnloadSeg()* on it.

Now look at how these calls to *UnloadSeg()* fit into a program:

```
void main( void )
{
   Initialize_Toolbox();
   Set_Up_Menu_Bar();

   while ( All_Done == FALSE )
   {
      Handle_One_Event();
      Unload_All_Segments();
   }
}

void  Unload_All_Segments( void )
{
   UnloadSeg( Do_Screen1 );
```

```
    UnloadSeg( Do_Screen2 );
    UnloadSeg( Do_Screen3 );
    UnloadSeg( Do_Screen4 );
    UnloadSeg( Initialize_Toolbox );
}
```

Lesson 9–2: Segmentation

You can run the program enclosed with this book for a hands-on tutorial about this topic.

Setting a Program's Size

When a user double-clicks on an application's icon in the Finder, the system sets up a memory partition for that application, then loads part or all of the program into the partition. The size of the application's partition is initially set up by the programmer but can be overridden by the user.

The user's role in setting the partition size

All programs come with a partition size suggested by the program's manufacturer. The program's user can change the partition size by selecting "Get Info" from the File menu in the Finder. In any version of System 6 the user can make just a single change to the partition size. That's shown on the right side of Figure 9–23.

With System 7 the user can set both a new *minimum partition* size and a *preferred size*. The minimum partition size is the limit below which the application will not run. The preferred partition size is the memory size at which the user feels the application can run more effectively. If the amount of memory entered in the preferred size is not available, the system will place the application into the largest available block of memory. Allowing the user to configure the partition size lets him base the program's partition on the amount of RAM installed in his Macintosh. The System 7 Get Info dialog box is shown on the left side of Figure 9–23.

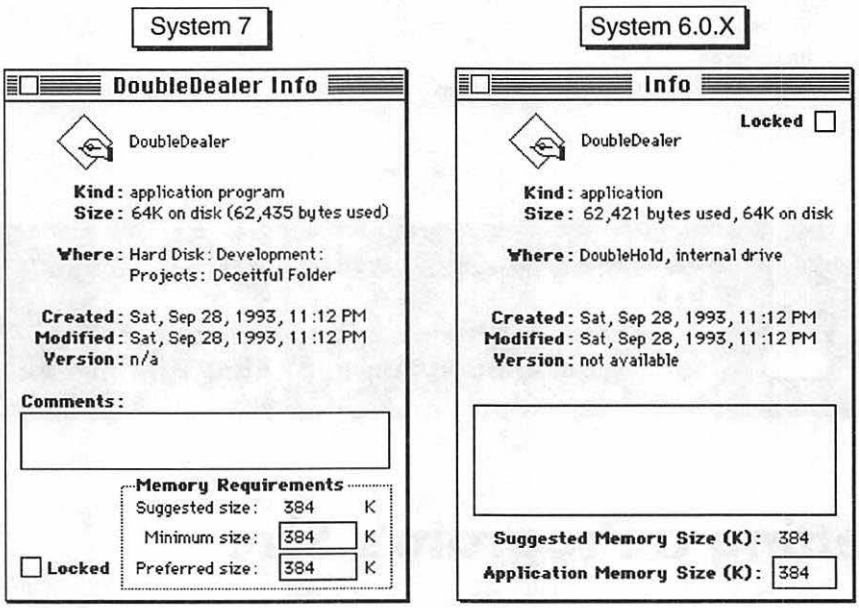


Figure 9–23. *Get Info dialogs under System 7 and System 6*

Memory chip prices have fallen greatly in recent years, and the amount of memory in users' Macintoshes is increasing. You may wonder if its worth the extra effort to plan out segmentation. After all, you could just do what many program manufacturers do and assign a very large partition to your program, guaranteeing that the entire program will load and stay in memory. That's exactly why you shouldn't. As Macs get more memory users are loading more of these large programs at once. A user, even one with 8 Mb of RAM or more, with several programs running at once might still often find himself just 100K shy of being able to load another program—maybe yours.

Setting an application's partition size in THINK C

You're the manufacturer of your program, so you get to set the manufacturer's suggested size for your program's partition. To do this in THINK C, you select "Set Project Type" from the Project menu before you build your application. The dialog box that appears is shown in 9–24.

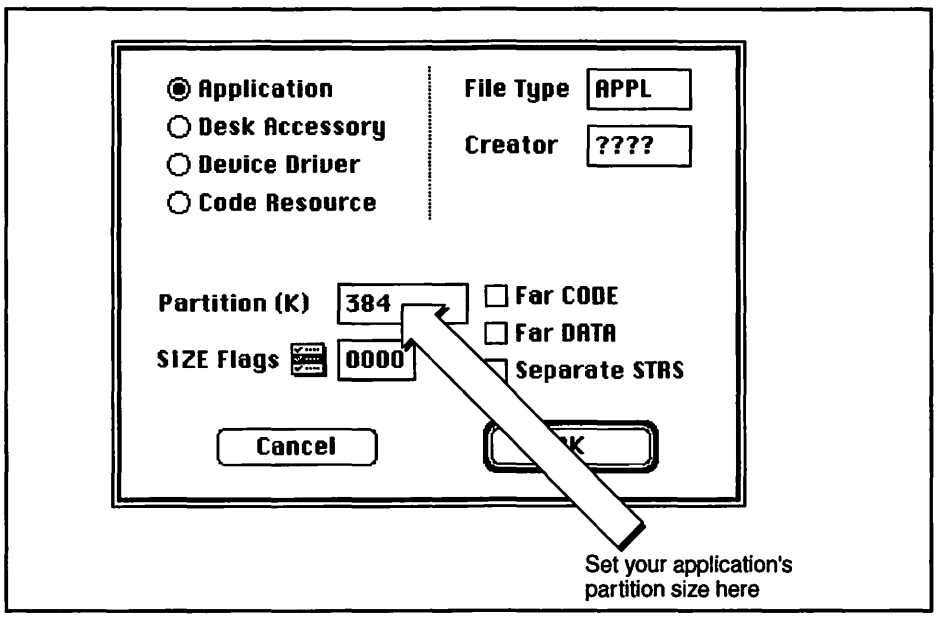


Figure 9–24. *Setting your application's partition size in THINK C*

If you don't specify a partition size, THINK C gives your program a size of 384 K. Whether you or THINK C set the size, the partition size will appear as the suggested, minimum, and preferred size in the Get Info dialog box of the Finder's File menu. That's shown back in Figure 9–23. To hold the partition size information, THINK C creates a 'SIZE' resource and adds it when it builds your application. If you want to prefill the minimum and preferred partition sizes with values other than those THINK C uses, you'll want to edit your application's 'SIZE' resource after building it.

Using the 'SIZE' resource to set a partition

When THINK C builds your final application, it adds a 'SIZE' resource with ID -1 to your application. The ID of the 'SIZE' is important. If one of your program's users makes any change to partition size using the "Get Info" menu option, a new 'SIZE' resource with an ID of 0 will be added to your program. Each time your program is launched the system looks at your program to see if it contains a 'SIZE' with an ID of 0. If the user made changes, the system will accept the user's values over those you've set. If no 'SIZE' with an ID of 0 is present, the system will then use the values you set up with an ID of -1.

When THINK C sets your program's suggested partition size, it also sets your program's preferred and minimum sizes to the same value. After building your program, run ResEdit and open your program. Don't open the program's resource file; open the program itself. There you'll see a 'SIZE' resource type. Double-click it to see the list of 'SIZE' resources. At this point there will be just one. Figure 9–25 shows what you'll see in ResEdit.

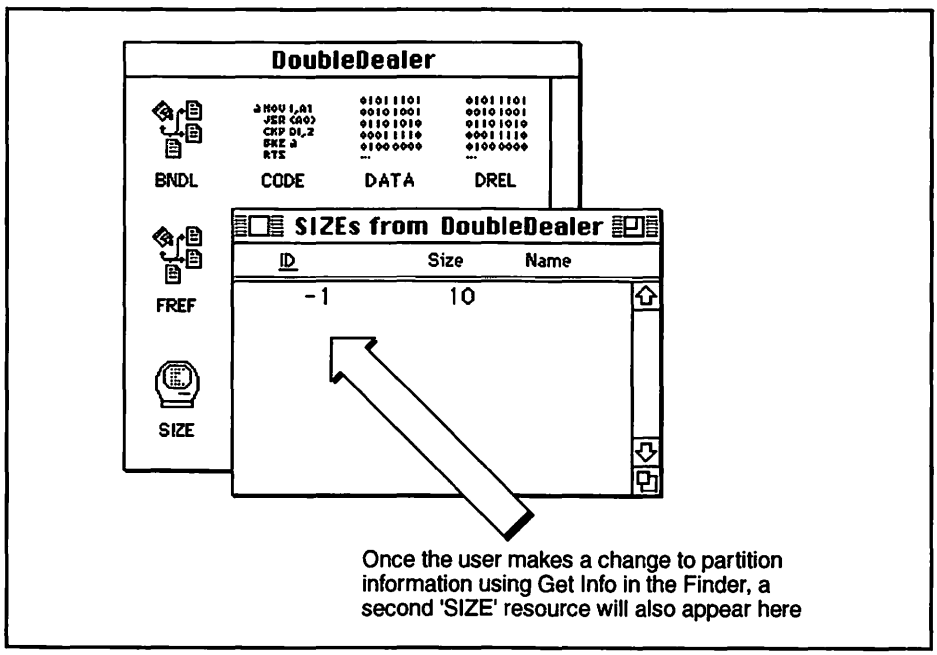


Figure 9–25. *The 'SIZE' resource in ResEdit*

Double-click on the 'SIZE' ID of -1 to edit it. When the size editor opens, scroll down to the bottom of the edit window. There you'll see two edit text boxes. They're shown in Figure 9–26. The value you enter in the top box will become the application's preferred partition size. This value is also used as the program's suggested size, so any change you make here will override the value you set in the THINK C environment. The value you type in the lower box will be the program's minimum size. In Figure 9–26, both values are set to 393,216 bytes—384 K.

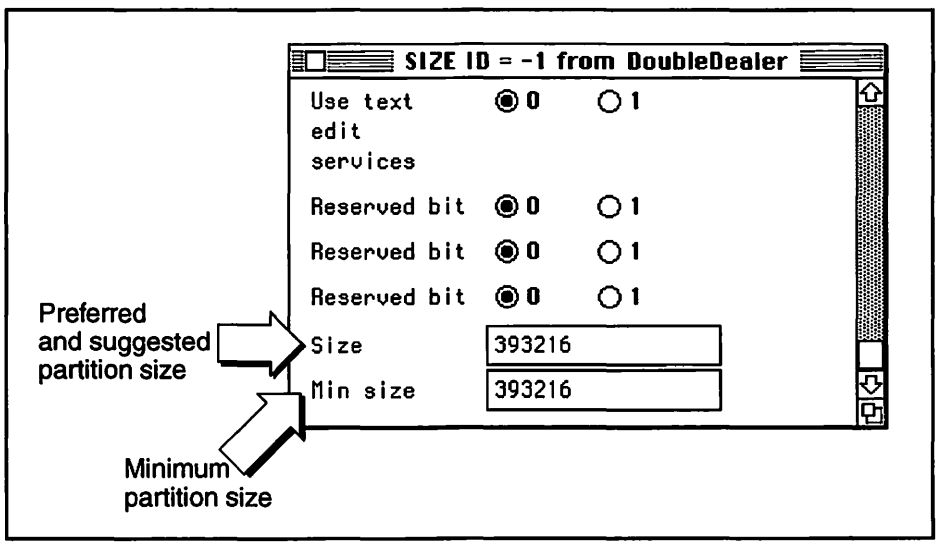


Figure 9–26. *Editing the 'SIZE' resource in ResEdit*

While you're in the 'SIZE' resource, scroll up a little. You'll see a pair of radio buttons labeled "32-bit Compatible". I discussed 32-bit clean programs earlier in this chapter. If you've thoroughly tested your program on a Macintosh that has 32-bit addressing turned on, set this radio button as shown in Figure 9–27. That lets the Finder know your program runs well in a 32-bit addressing environment.

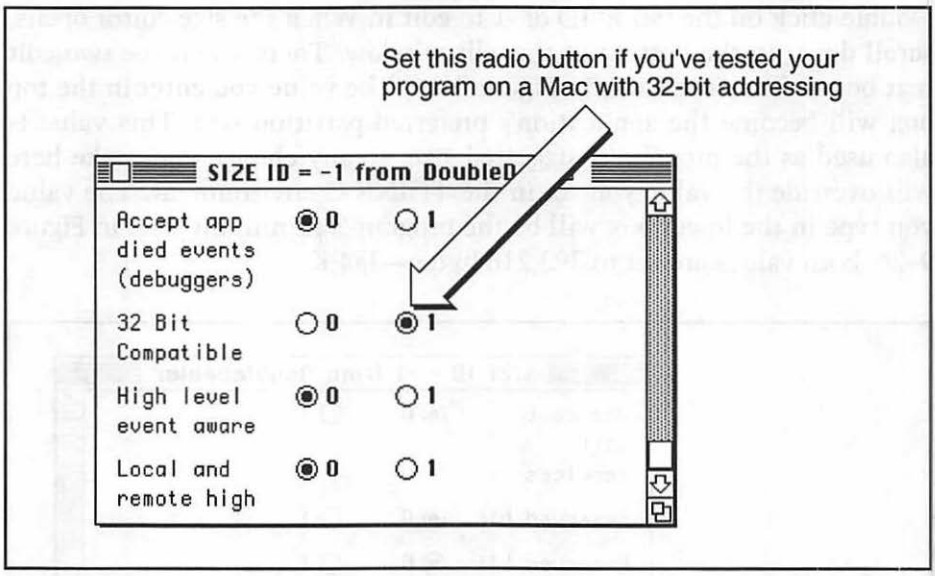


Figure 9–27. *Marking your program as 32-bit clean*

Determining your application's memory needs

Determining the memory requirements of your program is difficult. There are many factors that play a role in the amount of memory a program needs. Here are some of those factors:

- Loading of static 'CODE' resources, such as 'CODE' 1.
- Loading and unloading of purgeable 'CODE' resources.
- Creation of objects in response to program menu commands; this can vary based on user's selections.
- Amount of global data.
- Number of calls that take place between segments.
- Size of the stack.

Some factors you may be able to determine, including the amount of memory the static 'CODE' resources will occupy. If you are familiar with debuggers, you can use either MacsBug or TMON as a heap-exploring tool to help you determine the dynamic memory requirements of your program. I don't cover debuggers here; their use is a topic worthy of an entire book.

If you're planning on thumbing through every Macintosh book you can find in order to find a simple formula for the calculation of a program's partition size, save your time and energy. Such a book doesn't exist.

Many of you may be overwhelmed by the number of factors involved in determining memory use. And you may not be well versed in the use of debuggers. You may be wondering if there are any "quick and dirty" methods of getting at least a rough idea of program memory use. There are, and I cover two of them next.

Watching program memory using the Finder

If you're using THINK C, set your program's partition size as discussed earlier, then build your application. You can start with the default size of 384K that THINK C uses. Leave the THINK C environment. Go to your program's icon in the Finder and double-click on it to run your program.

Put your program through its paces. Select menu options, open dialogs, force the program to use the data structures you've programmed into it. In short, do everything the user will be allowed to do. And do each thing more than once.

As you're running your program, click periodically on the desktop. This will take you out of your program and into the Finder. The menu bar will change to that of the Finder. Select "About This Macintosh" from the Apple menu. You'll see a dialog box like that shown in Figure 9–28.

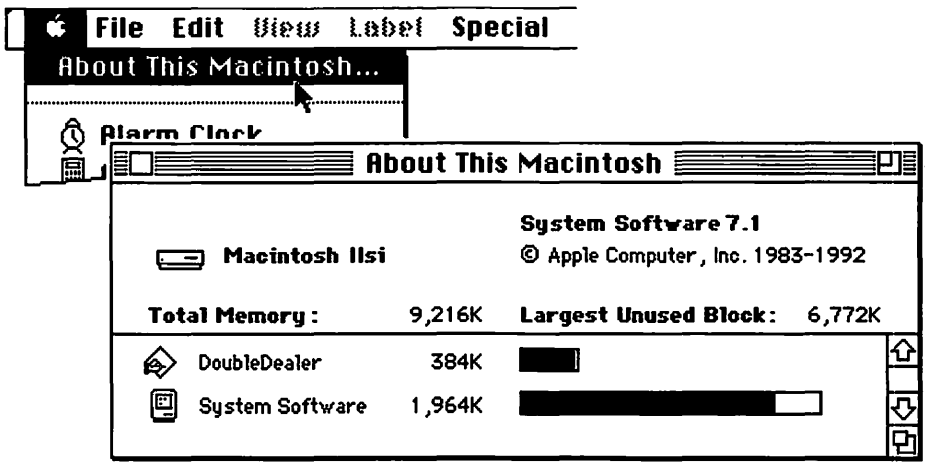


Figure 9–28. *The "About This Macintosh" dialog box*

The dialog box you see when you select "About This Macintosh" shows information about memory use for your Macintosh. The bar that displays your program's partition, and the amount of it that is currently in use, will be of most interest to you. This bar will fluctuate in length as your program runs. Figure 9–28 shows you that the *DoubleDealer* program has just about filled its 384 K partition at this point.

If you continue to run *DoubleDealer*, will the partition fill completely and crash our program? Maybe it will, and maybe it won't. Remember, memory allocation is dynamic in both directions—this program both frees memory by purging objects from memory and consumes memory by loading objects. The next action taken in *DoubleDealer* may cause one 'CODE' resource to be purged and a smaller one loaded. This would free up some of the memory in the application's partition.

In any case, *DoubleDealer* is reaching its partition limit. Quit the program and select "Get Info" from the Finder's File menu. Change the program's minimum and preferred sizes to a higher value, as shown in Figure 9–29.

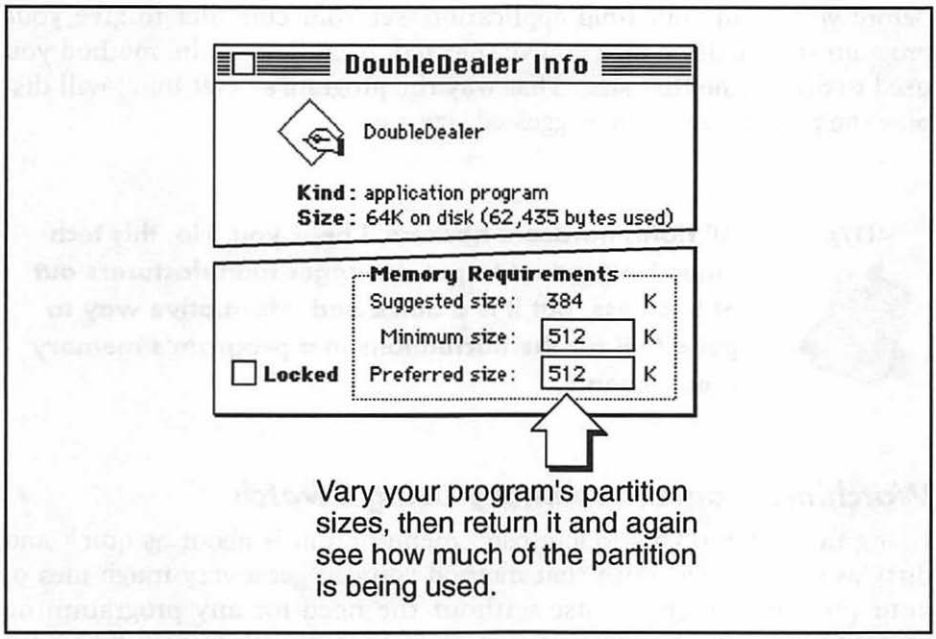


Figure 9–29. *Increasing a program's partition size during testing*

Now, run your program again. And test it vigorously. Check the About This Macintosh dialog box periodically. I did this with *DoubleDealer*, and the most memory use I saw is shown in Figure 9–30. For *DoubleDealer*, the 512 K partition seems more appropriate than the 384 K partition I started with.

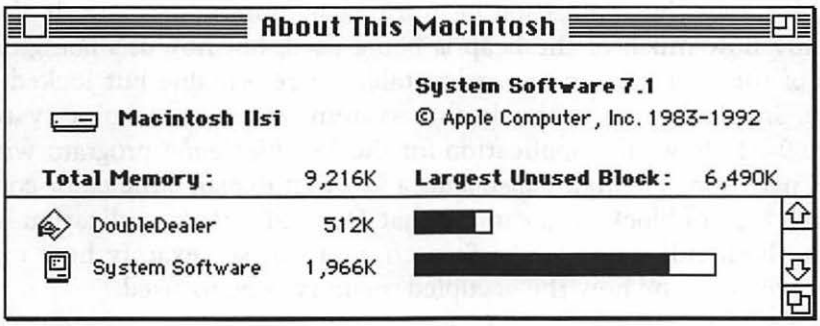


Figure 9–30. *The About This Macintosh dialog box after a program partition size change*

Before you build your final application, set your compiler to give your program the partition size you've selected, regardless of the method you used to determine that size. That way the program's "Get Info" will display the proper size in the suggested size area.

All right, hardcore hackers, I hear you. No, this technique isn't meant to put debugger manufacturers out of business. But it is a quick and informative way to get a feel for the fluctuations in a program's memory requirements.

Watching program memory using Swatch

Using the "About This Macintosh" menu option is about as quick and dirty as you can get. With that method you can get a very rough idea of your program's memory use without the need for any programming tools. To get a much more accurate idea of what's going on in RAM, try running a nifty utility program called *Swatch*. In fact, *Swatch*, written by Joe Holt, is so handy I've put a copy of it on the disk that is included with this book. *Swatch* is a very small Macintosh program (40K) that has just one purpose: it watches the memory usage of all applications that are running. The window that *Swatch* displays, shown in Figure 9–31, gives much more information than the window you see using About This Macintosh.

Swatch shows the application heap for each running program. It shows not only how much of the heap is being used, but how it's being used. Parts of the heap that are nonrelocatable, or relocatable but locked, are shown in black on a monochrome system or red on a color system. Figure 9–31 shows the application for the *DoubleDealer* program with a 384 K partition. The figure also adds a key that explains the color-coding for each type of block. You can see that *DoubleDealer*'s application heap is just about full. But here in *Swatch*, you can see exactly how much memory is free and how the occupied memory is being used.

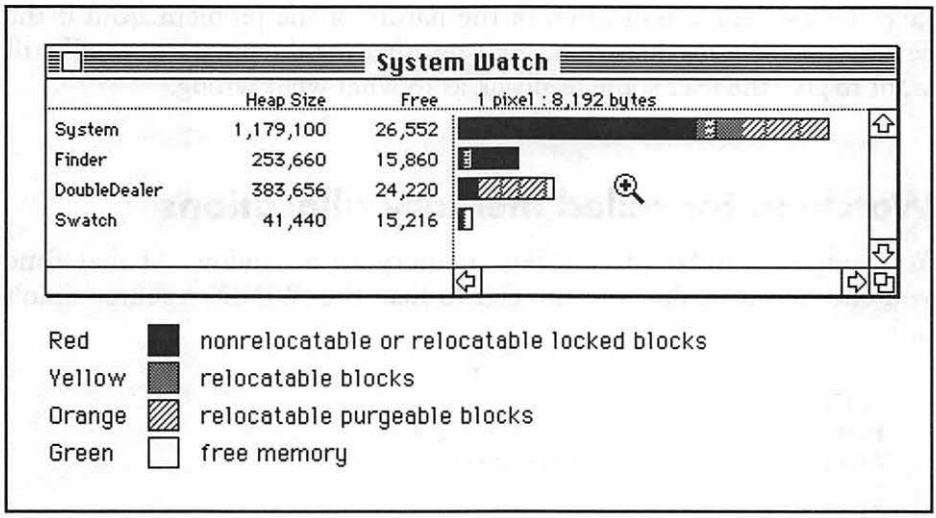


Figure 9-31. *The Swatch window, as Swatch views the system*

Notice in the above figure that the cursor has the appearance of a magnifying glass with a cross in it. By clicking the mouse you can magnify the right side of the window to get a more detailed view of memory. As shown, one pixel represents 8,192 bytes of RAM. A click of the mouse will make one pixel represent only 4,096 bytes. You can keep clicking to get more and more detail. Holding the Option key while clicking the mouse will reduce the view. *Swatch* has a few other tricks that provide more insight into the memory your program and they're mentioned in the text file included on disk in the *Swatch* folder. The text file also lists the few steps involved in copying the program to your disk.

Computer memory is an abstract concept that lends itself to much confusion for both beginner and advanced programmer. *Swatch*'s ability to allow you to visualize memory helps clarify just what's going on in those mysterious RAM chips of the Macintosh.

Handling Memory Errors

Even with careful planning, your program could fail at some point. If the error is severe, such as failed memory allocation for a window necessary to the execution of your program, you'll want to exit the program gracefully rather than have the screen freeze. Before you do so, you'll want to

give the user some indication of the nature of the problem. And if the severity of the error doesn't warrant terminating the program, you'll still want to give the user some feedback as to what went wrong.

Watching for failed memory allocations

You've used *NewPtr()* to allocate memory for a window. At that time you checked to make sure the call to load the 'WIND' resource didn't fail:

```
#define     NIL                       0L
#define     WIND_ID                 128
#define     IN_FRONT     (WindowPtr)-1L

WindowPtr   new_window;
Ptr         wind_storage;

wind_storage = NewPtr( sizeof( WindowRecord ) );

new_window  = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
if ( new_window == NIL )
   ExitToShell();
```

You should also check to confirm that the memory you attempted to allocate was indeed allocated.

```
wind_storage = NewPtr( sizeof( WindowRecord ) );
if ( wind_storage == NIL )
   ExitToShell();

new_window  = GetNewWindow( WIND_ID, wind_storage, IN_FRONT );
if ( new_window == NIL )
   ExitToShell();
```

The first *if* statement checks to see that memory for the window is available. If the program reaches the second *if* statement, memory is indeed available. What then could cause a call to *GetNewWindow()* to fail? A 'WIND' resource with *WIND_ID* doesn't exit. This is unlikely, but could occur if the user opened your program with ResEdit "just to look around" and instead deleted or renumbered some resources unintentionally.

The solution here to handling a memory problem has been to call *ExitToShell()*. Bailing out of the program and returning the user to the Finder, rather than risking a frozen Mac, is thoughtful. But the user will appreciate it much more if you first provide him with a hint about the problem. That gives the user the chance to correct the problem and try again.

Providing the user with error information

Before exiting a program after a severe error, call an error handling routine of your own creation. Pass the routine a number that represents the type of error that occurred. Also, send the routine a flag that indicates if the nature of the error warrants termination of the program.

```
#define   ERR_WIND_MEM_ALLOCATE_FAIL      1

#define   DO_TERMINATE_ERROR           TRUE
#define   DONT_TERMINATE_ERROR        FALSE

wind_storage = NewPtr( sizeof( WindowRecord ) );
if ( wind_storage == NIL )
   Post_Error_Message( ERR_WIND_MEM_ALLOCATE_FAIL, DO_TERMINATE_ERROR );
```

The error handling routine should post an alert that displays a message appropriate to the error that occurred. The displayed message will be based on the first parameter passed to it. Whether the program should be terminated will depend on the value of the second parameter. Here's a typical error-handling routine:

```
#define   ERR_WIND_MEM_ALLOCATE_FAIL      1
#define   ERR_PICT_MEM_ALLOCATE_FAIL      2
[ have a #define for each error condition you look for ]

#define   DO_TERMINATE_ERROR           TRUE
#define   DONT_TERMINATE_ERROR        FALSE

#define   ERR_STR_LIST                  128
#define   ERR_ALRT_ID                   128

void  Post_Error_Message( short error_num, Boolean terminate )
{
```

```
    Str255  the_str;

    GetIndString( the_str, ERR_STR_LIST, error_num );
    ParamText( the_str, "\p", "\p", "\p" );
    StopAlert( ERR_ALRT_ID, NIL );

    if ( terminate == DO_TERMINATE_ERROR )
       ExitToShell();
}
```

You can keep the error-handling routine nice and short, as I've done, by keeping all the error messages in a 'STR#' list. The first parameter passed to the routine will be an index into the list. Call *GetIndString()* to dig the string out of the 'STR#' list. (This technique was first covered back in Chapter 3.) Figure 9–32 shows a typical 'STR#' resource for error messages.

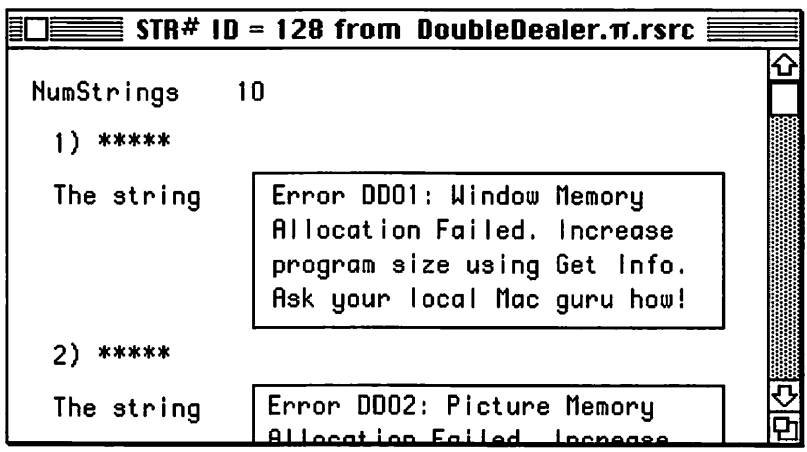


Figure 9–32. *'STR#' list of error messages*

You can give each error message a reference number, such as *DoubleDealer*'s DD01, DD02, and so forth. That will help when your program becomes a smash hit and you institute a technical support phone number! When a user calls your technical support number all he has to report is the error number, and your technicians can look up the error in your Error Reference Journal to get more information about it.

After retrieving the error string from the 'STR#' list, use *ParamText()* to set the first of four strings to this message string. That's the string that will be displayed in the alert. Figure 9–33 reminds you how to use the

"^0" notation in the 'DITL' of an 'ALRT' resource, as introduced in Chapter 7. Figure 9–34 shows the alert the user will see.

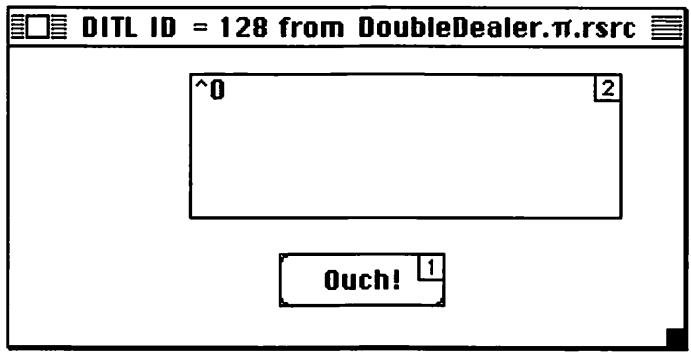


Figure 9–33. *'DITL' to display one* ParamText() *string*

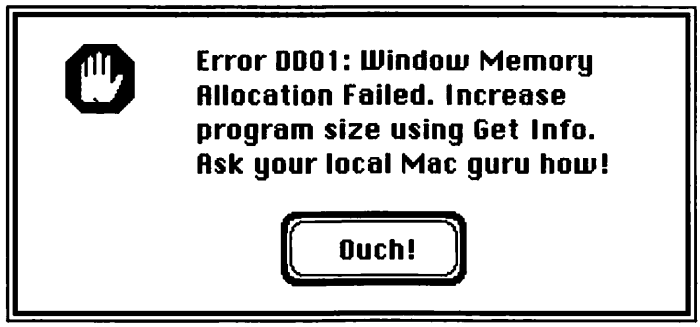


Figure 9–34. *The error message alert*

Numerous conditions can cause problems in a program: lack of available memory is one of the most common. You'll want to include error checking where appropriate in your particular application.

Chapter Program: Tying it All Together

This chapter's example program, *InnerViewII*, is a more powerful version of last chapter's *InnerView* application. Many of the changes to the pro-

gram occur behind the scenes, and some are right up front for the user to see. The following are a few of the up-front changes to the program:

- The program now opens a modeless dialog that allows the user to select the type of machine features to display: hardware or software.
- The window that displays the results of the machine examination now displays either hardware or software information, based on the user's selection.
- Keyboard equivalents are used for menu items.
- An edit menu has been added to support editing in the information dialog's edit text item.

Here are a few of the things *InnerViewII* does behind the scenes:

- Stores all text displayed to a window in 'STR#' resources; no strings are hard-coded into the source code.
- Reserves memory for the informational dialog and results window early in the program.
- Performs error-checking at critical points in the program.
- Uses multiple source files and header files in the THINK C project.
- Uses segmentation, as described in the chapter.

Figure 9–35 shows both the dialog and window that InnerViewII displays.

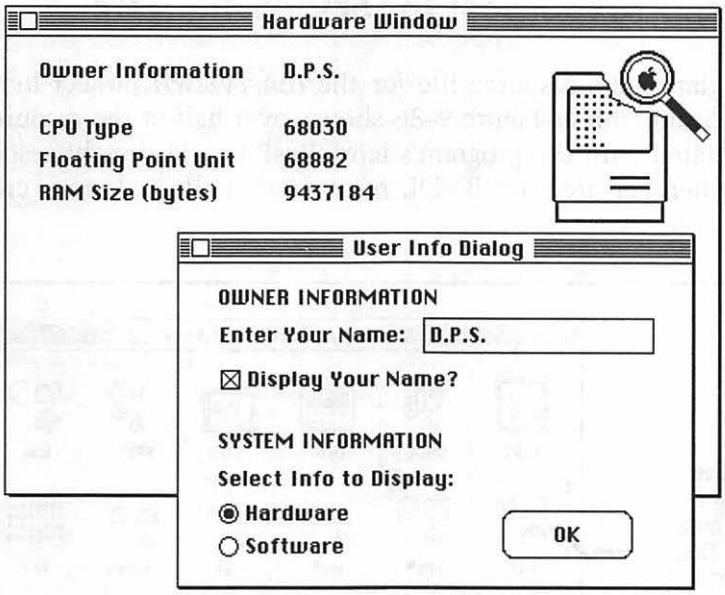


Figure 9-35. *A look at* InnerViewII

The results window displays information about three environment factors in the program user's Macintosh. I've sized the window so that a person knowledgeable about the *Gestalt()* function can add about eight more. As a matter of fact, if a crafty programmer removed the picture, reduced the spacing between lines, and used two columns, that programmer could probably fit about two dozen pieces of information in the window.

Hint, hint! Yes, that's you! You have the project, the source code, and the know how. *InnerViewII* is a good base for a useful utility program. If you instruct the program user to place your version of *InnerViewII* in the Apple Menu Items folder in the System Folder, it can be accessed from the Apple menu. Any time a user has a question about what's in his Mac, he can simply run your program from the Apple menu.

Program resources: InnerViewII.π.rsrc

At first glance the resource file for the *InnerViewII* project looks a bit overwhelming. But as Figure 9–36 shows, over half of the resource types are associated with the program's icon. ResEdit added eight resources to the file when I created the 'BNDL' resource that allowed me to create and edit the icon.

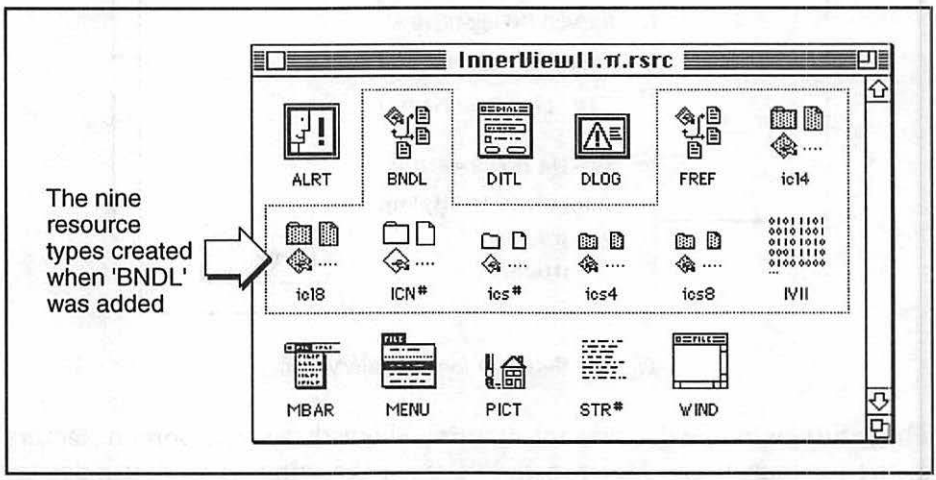


Figure 9–36. InnerViewII.π.rsrc *file*

Figure 9–37 shows the 'BNDL' resource. The six icons pictured are each separate resources. I gave the 'BNDL' a signature of "IVII" for *InnerViewII*. When you get to the THINK C project, you'll use these four same characters to set the program's creator. Chapter 3 covered program icons, the 'BNDL' resource, and signature and creator.

The *InnerViewII* program displays a window with the results of the program's check for machine features. Whether the user elects to see hardware or software features, the program uses the same 'WIND' resource.

InnerViewII has one 'DLOG' resource and a companion 'DITL' for the program's modeless information dialog box. Figure 9–38 shows the items and item numbers for this dialog box.

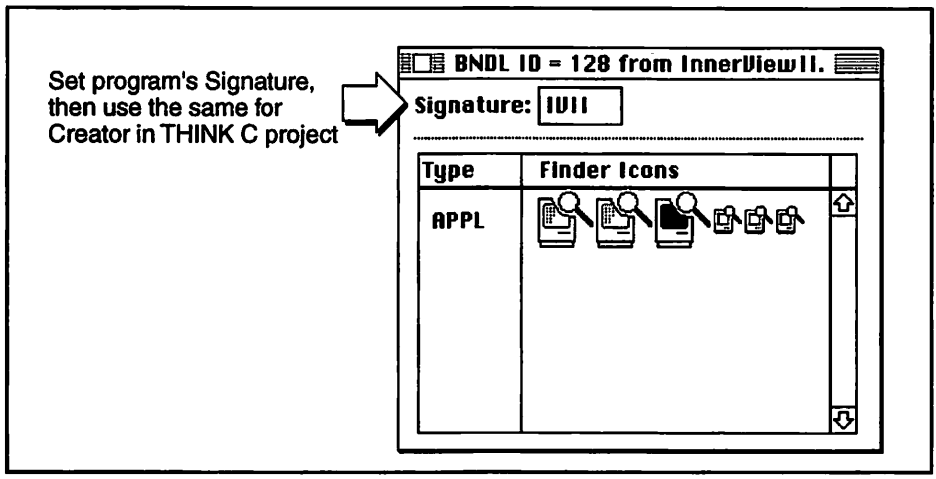


Figure 9–37. *The 'BNDL' resource for* InnerViewII

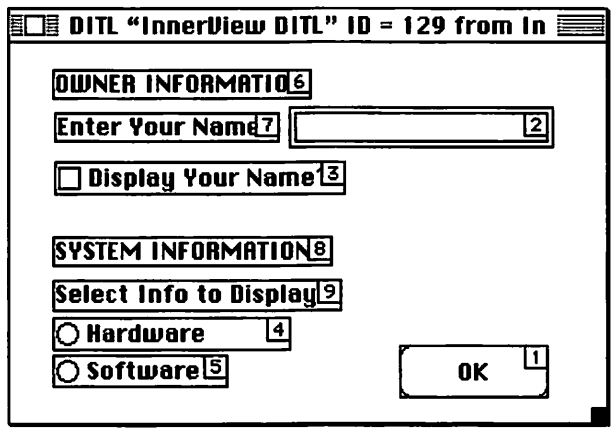


Figure 9–38. *'DITL' resource for* InnerViewII

The program has two other 'DITL' resources. One corresponds to the 'ALRT' that *InnerViewII* puts up in response to a menu choice of "About *InnerViewII*" from the Apple menu. Chapter 6 covered alerts.

An 'ALRT' uses the other 'DITL' to display error messages. *InnerViewII* uses the error-handling method discussed in this chapter.

The results window of the *InnerViewII* program displays the same picture as last chapter's program; they're repeated here as Figure 9–39. The resource

file holds two versions of the picture: one in color and one in monochrome. Chapter 8 covered the reasons for storing two versions of a picture.

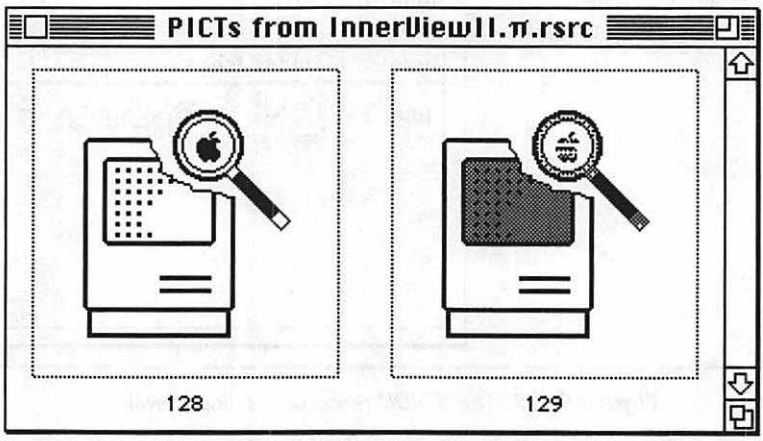


Figure 9–39. InnerViewII *'PICT's*

Figure 9–40 shows the program's three 'MENU' resources. Each of the File menu items has a keyboard equivalent, as do the enabled items in the Edit menu. *InnerViewII* doesn't support Undo or Clear, so these items have been marked as disabled here in the resource file. Why include them in the menu? Uniformity. Mac users expect to see five standard editing commands in the Edit menu of a Macintosh program, so I include them here.

The three 'MENU' resource IDs appear in the program's one 'MBAR' resource. Chapter 7 covered these two resource types.

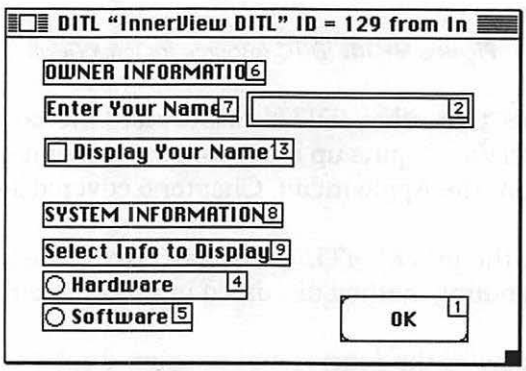


Figure 9–40. *'MENU' items for* InnerViewII

The last resource type is the 'STR#', and it has plenty of items. Figure 9–40 shows the 11 'STR#' resources. All but one of the 11 contain more than one string.

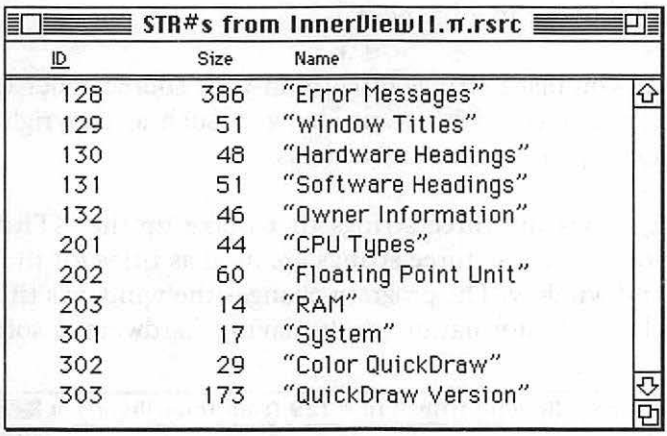


Figure 9–41. *'STR#' resources for* InnerViewII

Apple strongly suggests that any text you display in a window or dialog be stored as a resource. Why? If a change becomes necessary, text in a resource is much easier to change than text embedded within source code. If the text is hard-coded into the source code, the person who wants to make a change must also have the source code. After the change, the source code must be recompiled into a new application.

If text is placed in strings within 'STR#' resources, anyone with a working knowledge of ResEdit can make text changes to your program without access to your source code. Someone can use ResEdit to open the application and edit the strings in the resources of the program itself.

Why is it a good idea to allow this easy access to program text? Here are a few reasons.

- It's easier for users of your program to change wording or make corrections of typos, with your permission, of course.
- It's easier to translate your program to another language, again with your permission.

How likely is it that your program will be used in foreign countries? It may not be as improbable as you think. And, you may want to cater to the large non-English speaking segment of the USA. A translator, even one with very little knowledge of programming, can make the necessary changes without your involvement.

When should you place strings directly in your source code? Only when you have text that you don't want changed, such as copyright information or your company's name and address.

Figure 9–42 shows the three strings that make up the 'STR#' resource with an ID of 129. These three strings are used as titles for the program's dialog box and window. The program changes the window's title, depending on which set of information it's displaying: hardware or software.

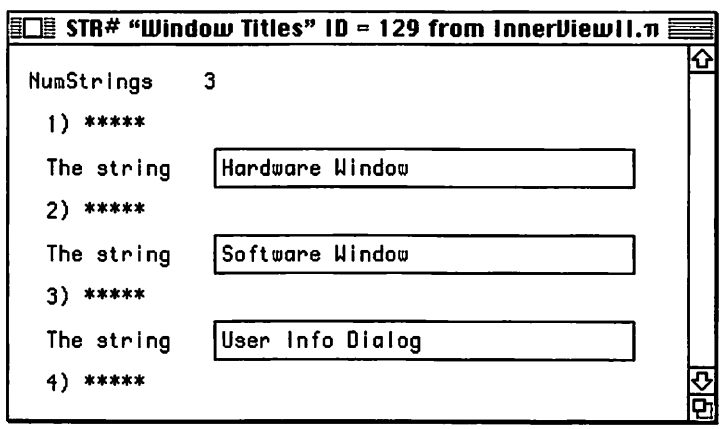


Figure 9–42. *The three strings in 'STR#' 129*

A second example of how *InnerViewII* uses 'STR#'s appears in Figure 9–43. The *InnerViewII* window displays titles for the three pieces of information being retrieved. Rather than placing "\p" strings directly in the source code (as last chapter's *InnerView* did) the new InnerViewII keeps the titles in a 'STR#' resource. The program calls *GetIndString()* to retrieve the strings and then draw them to the window. Figure 9–44 demonstrates how some of these 'STR#'s are used by the program.

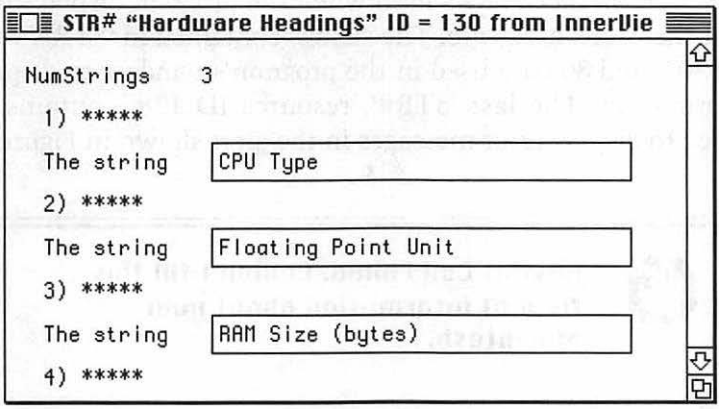


Figure 9–43. *The three strings in 'STR#' 130*

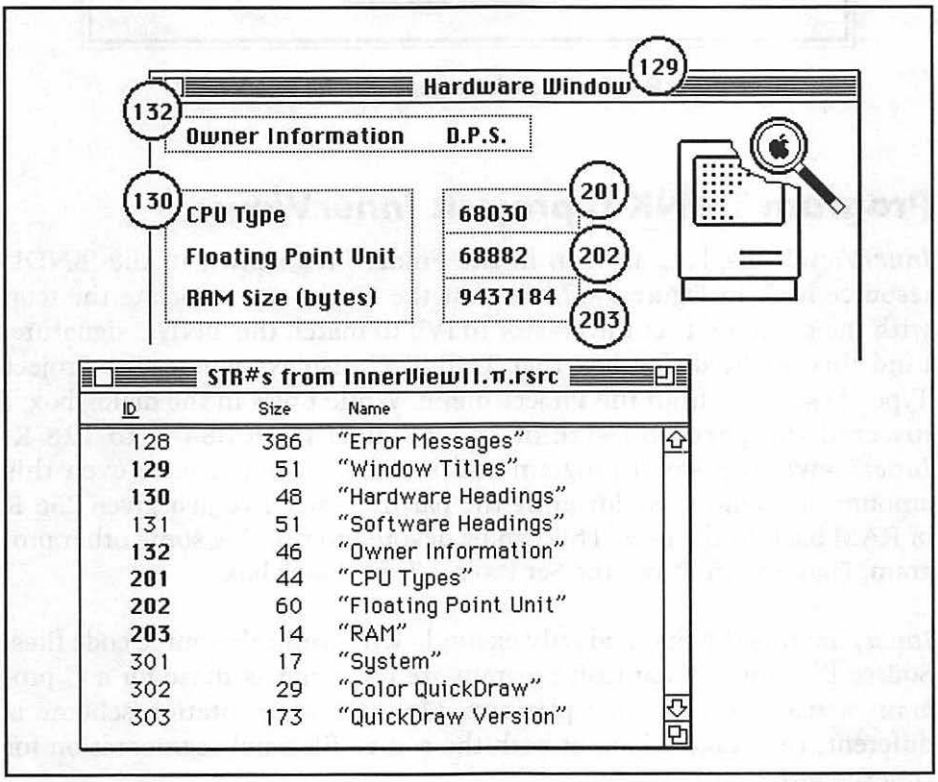


Figure 9–44. *How* InnerViewII *uses the 'STR#' resources*

Figure 9–44 shows the 'STR#'s used when the program displays information about the user's hardware. The strings contained in 'STR#' resources 131, 301, 302, and 303 are used in the program's window to display software information. The last 'STR#', resource ID 128, contains several strings used to display error messages in the alert shown in Figure 9–45.

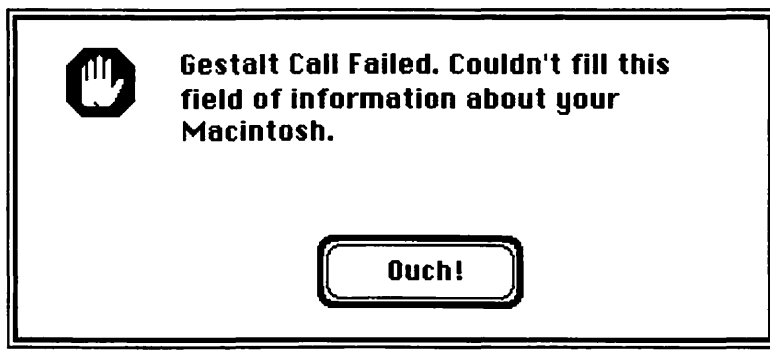


Figure 9–45. *The error message alert for* InnerViewII

Program THINK C project: *InnerView.π*

InnerViewII displays an icon in the Finder. (It's shown in the 'BNDL' resource back in Figure 9–37.) So that the Finder can associate the icon with the program, I set the creator to IVII to match the 'BNDL' signature. I did this in the dialog box that THINK C displays when "Set Project Type" is selected from the Project menu. While I was in the dialog box, I lowered the partition size of *InnerViewII* from 384 K to 128 K. *InnerViewII* is a small program and probably won't require even this amount of memory. By lowering the partition size I've just given 256 K of RAM back to the user. This can be devoted to running some other program. Figure 9–46 shows the Set Project Type dialog box.

InnerViewII is the first and only example with multiple source code files. Source files for a Macintosh program are the same as those for a C program written on any other platform. Only the segmentation scheme is different. Let's take a look at both the source files and segmentation for *InnerViewII*.

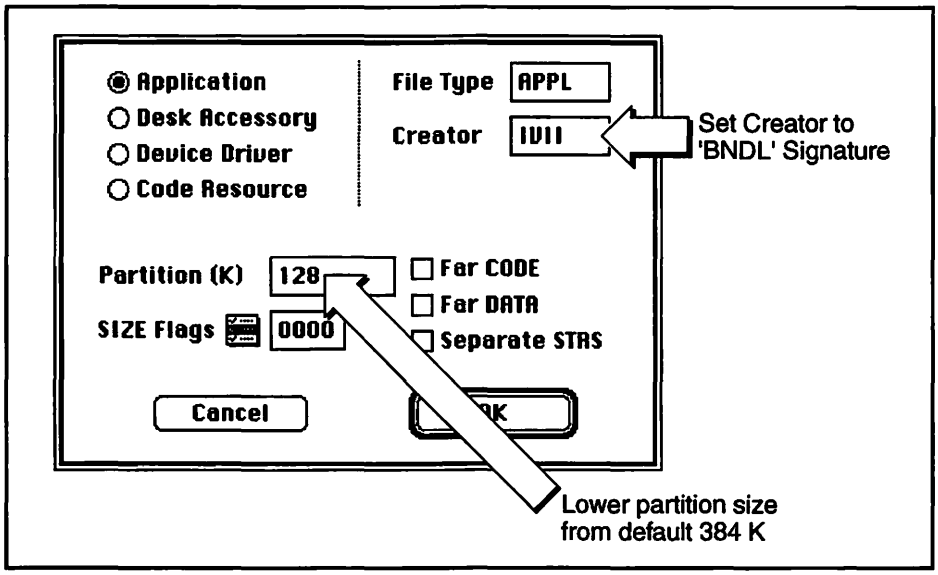


Figure 9–46. *Setting creator and changing partition size in THINK C*

The code for *InnerViewII* is divided between three source files: *InnerViewII.c*, *Initialize.c*, and *Utilities.c*. All of the global variables are in a header file, *Globals.h*. All of the *#define* directives appear in their own header file, *Defines.h*. Figure 9–47 shows the relationship between these files.

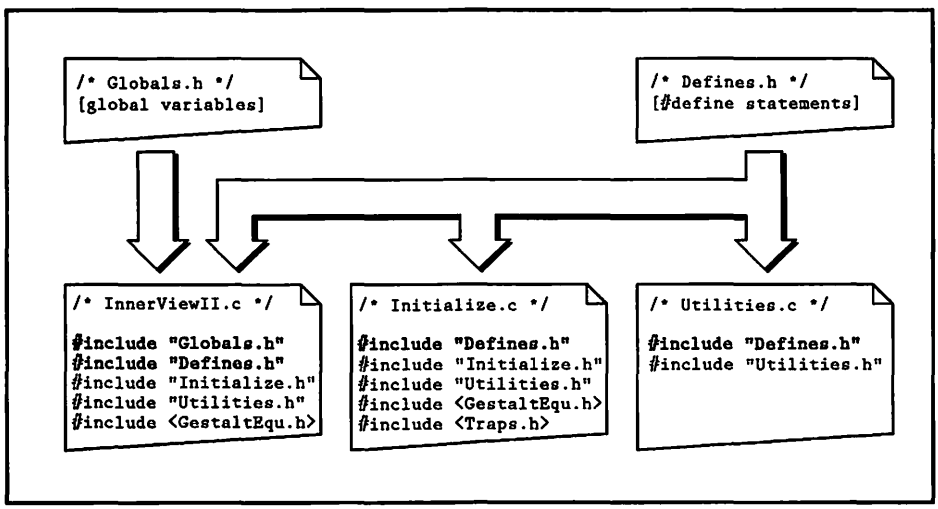


Figure 9–47. *Source file relationships to globals and defines*

Notice in Figure 9–47 that only *InnerViewII.c* includes the *Globals.h* header file. *Globals.h* contains all of the global variable declarations, and can therefore only be a *#include* for one source file. Otherwise the compiler views the situation as multiple declarations of the same variable. Any global variables used by either *Initialize.c* or *Utilities.c* have to be declared within those files using the "extern" keyword. I show that in the source code listings that follow this section.

All three source files include the *Defines.h* header. This header contains only *#define* directives. The compiler simply uses *#defines* to make substitutions in your source code; it doesn't use them as declarations.

The application has two defined header files, *Initialize.h* and *Utilities.h*. They contain public interfaces to the routines in *Initialize.c* and *Utilities.c*. All of the source files need to know about the routines in *Utilities.c*, so they all include *Utilities.h*. None of the routines in *Utilities.c* calls routines in *Initialize.c*, so you won't find a *#include* directive for it in *Utilities.c*. This is illustrated in Figure 9–48.

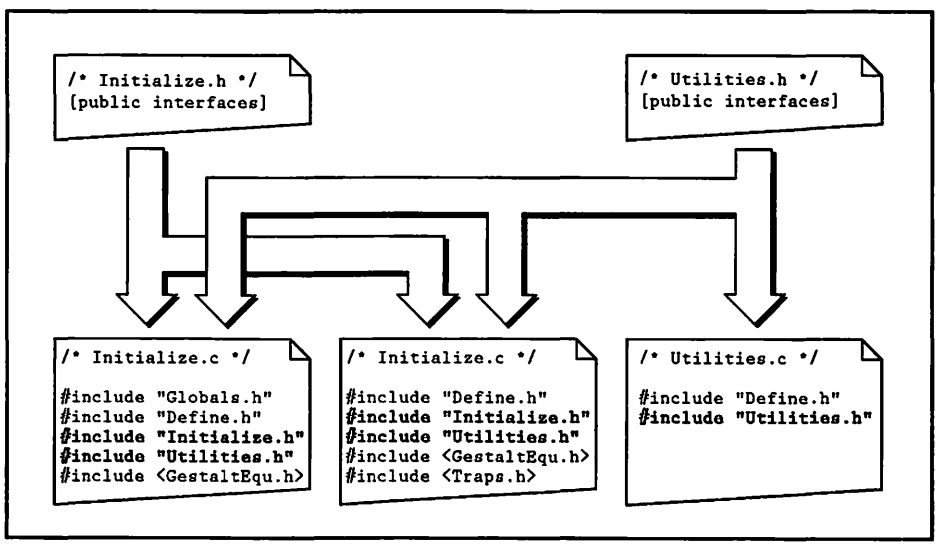


Figure 9–48. *Source file relationships to application-defined headers*

Two Apple header files, *Traps.h* and *GestaltEqu.h* support calls to *NGetTrapAddress()* and *Gestalt()*. Both *InnerViewII.c* and Initialize.c contain functions that call *Gestalt()*, so they include *Gestalt.h*.

Initialize.c also includes *Traps.h*. The *Utilities.c* source file doesn't need either of the Apple includes. This is summarized in Figure 9–49.

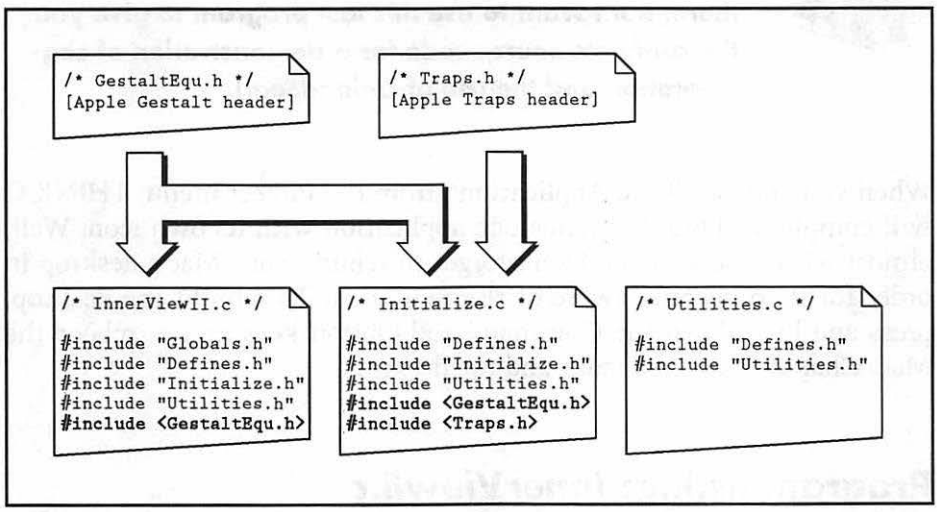


Figure 9–49. *Source file relationships to Apple-defined headers*

The *InnerViewII.π* project has three segments. I've kept the bulk of the code in Segment 2, which I've renamed "Main". This is shown in Figure 9–50. I also double-clicked on the segment name to change its attributes to preloaded and locked. That puts it at the base of the application's partition. (This technique was explained earlier in this chapter.)

I put all of the initialization routines in a file, *Initialize.c*, and then put that file in a separate segment—Segment 3. I did the same with utility routines; they're in *Utilities.c* in Segment 4.

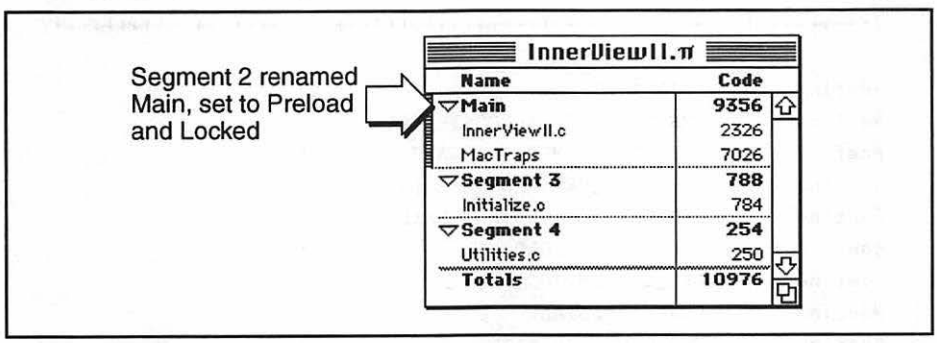


Figure 9–50. *Segmentation of* InnerViewII

How practical is it to segment about 10 K of source code, as done in *InnerViewII*? Couldn't you just have kept it all in one segment? All right, you caught me there. But I want to use this last program to give you the complete source code for a demonstration of segmentation and the use of *UnloadSeg()*.

When you choose "Build Application" from the Project menu, THINK C will compile and build a standalone application with its own icon. Well, almost with its own icon. Don't forget to rebuild your Mac's desktop in order for it to become aware of the new icon. To rebuild the desktop, press and hold down the Command and Option keys as you reboot the Mac. Chapter 3 covered icons and the desktop.

Program listing: *InnerViewII.c*

The *InnerViewII* program starts out with three source files and four application-defined header files, presented in their entirety now. If you're familiar with header files, you can just skim them here. A description of each file follows, with emphasis, on the new material.

Defines.h

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Defines.h                                                  */
/*   Purpose:  Define all constants.                                       */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#define    ERR_STR_LIST                    128
#define       ERR_WIND_MEM_ALLOCATE_FAIL     1
#define       ERR_DIALOG_MEM_ALLOCATE_FAIL   2
#define       ERR_PICT_MEM_ALLOCATE_FAIL     3
#define       ERR_CALL_TO_GESTALT_FAIL       4
#define       ERR_ROM_TOO_OLD                5
#define       ERR_SYSTEM_TOO_OLD             6
#define       ERR_NO_GESTALT                 7
#define       ERR_MENU_BAR_FAIL              8
```

```
#define    WIND_TITLE_STR_LIST              129
#define       WIND_HARDWARE_TITLE             1
#define       WIND_SOFTWARE_TITLE             2
#define       DLOG_INFO_TITLE                 3
#define    HARDWARE_TITLES_STR_LIST         130
#define    SOFTWARE_TITLES_STR_LIST         131
#define    OWNER_INFO_STR_LIST              132
#define       OWNER_TITLE                     1
#define       OWNER_NOT_AVAILABLE             2
#define    CPU_TYPE_STR_LIST                201
#define    FPU_TYPE_STR_LIST                202
#define    RAM_STR_LIST                     203
#define    SYSTEM_STR_LIST                  301
#define    COLOR_QUICKDRAW_STR_LIST         302
#define    QUICKDRAW_VER_STR_LIST           303

#define    IV_DLOG_ID                       129
#define       OK_BUTTON_DITL_ITEM             1
#define       NAME_DITL_ITEM                  2
#define       CHECK_DITL_ITEM                 3
#define       HARDWARE_DITL_ITEM              4
#define       SOFTWARE_DITL_ITEM              5
#define    DLOG_WIDTH                       320
#define    DLOG_HEIGHT                      200

#define    IV_WIND_ID                       128
#define    WIND_WIDTH                       485
#define    WIND_HEIGHT                      280

#define    ABOUT_ALRT_ID                    128
#define    ERR_ALRT_ID                      130

#define    MENU_BAR_ID                      128
#define    APPLE_MENU_ID                    128
#define       ABOUT_ITEM                      1
#define    FILE_MENU_ID                     129
#define       NEW_ITEM                        1
#define       TO_FRONT_ITEM                   2
#define       QUIT_ITEM                       4
#define    EDIT_MENU_ID                     130
#define       CUT_ITEM                        3
#define       COPY_ITEM                       4
#define       PASTE_ITEM                      5

#define    MAC_PICT_BW_ID                   128
```

```
#define    MAC_PICT_COLOR_ID                129

#define    NIL                                0L
#define    IN_FRONT              (WindowPtr)-1L
#define    REMOVE_EVENTS                       0
#define    SLEEP_TICKS                        0L
#define    MOUSE_REGION                       0L
#define    MENU_BAR_HEIGHT                    18
#define    DRAG_EDGE                          20
#define    CONTROL_ON                          1
#define    CONTROL_OFF                         0
#define    PIXEL_DEPTH_BW                      1
#define    PIXEL_DEPTH_MAX_COLOR              24
#define    ENTIRE_MENU                         0
#define    DO_TERMINATE_ERROR               TRUE
#define    DONT_TERMINATE_ERROR            FALSE
#define    GESTALT_ERR_TYPE                  999

#define    NUM_SOFTWARE_HEADINGS               3
#define    NUM_HARDWARE_HEADINGS               3
#define    LINE_HEIGHT                        20
#define    HEADING_X                          20
#define    HEADING_Y                          40
#define    RESULT_X                          170
#define    OWNER_Y                            25
#define    PICT_L                            380
#define    PICT_T                              5
#define    ASCII_ZERO                         48
```

Globals.h

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Globals.h                                                */
/*   Purpose:  Declare all global variables.                             */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

Boolean      All_Done = FALSE;
Boolean      Multifinder_Present;
EventRecord  The_Event;
MenuHandle   Apple_Menu;
MenuHandle   File_Menu;
MenuHandle   Edit_Menu;
```

```
Rect          Drag_Rect;
Point         Screen_Center;
WindowPtr     IV_Window_Ptr;
DialogPtr     IV_Dialog_Ptr;
Str255        Name_Str;
short         Old_Button_Num;
Boolean       Print_Name;
Boolean       Display_Hardware_Flag;
Boolean       Color_QD_Present;
short         Min_Pixel_Depth;
Ptr           Info_Dialog_Storage;
Ptr           Display_Window_Storage;
Str255        Mac_RAM_Str;
Boolean       Mac_RAM_Str_Error;
long          Mac_CPU;
long          Mac_FPU;
Str255        Mac_Sys_Str;
Boolean       Mac_Sys_Str_Error;
long          Mac_Has_Color_QD;
long          Mac_QD_Version;
```

Initialize.h

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Initialize.h                                            */
/*   Purpose:  Public interfaces for routines in Initialize.c.  */
/*             Make Initialize.c routines known to other files. */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void  Initialize_Toolbox( void );
void  Check_System( void );
void  Reserve_Window_Memory( void );
void  Initialize_Variables( void );
void  Set_Window_Drag_Boundaries( void );
void  Set_Screen_Center( void );
void  Set_Up_Menu_Bar( void );
void  Open_InnerView_Window( void );
void  Open_InnerView_Dialog( void );
```

Utilities.h

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Utilities.h                                                */
/*   Purpose:  Public interfaces for routines in Utilities.c.              */
/*             Make Utilities.c routines known to other files.             */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void  Post_Error_Message( short, Boolean );
void  Set_Check_Box( DialogPtr, short );
void  Set_Radio_Buttons( DialogPtr, short );
void  Get_Text_From_Edit( DialogPtr, short, Str255 );
```

Initialize.c

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Initialize.c                                               */
/*   Purpose:  Initialization routines and other functions that            */
/*             are "one-time-only."                                       */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/


/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include "Defines.h"
#include "Initialize.h"
#include "Utilities.h"
#include <Traps.h>
#include <GestaltEqu.h>

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++  Make file aware of externally-declared global variables ++*/

extern  Boolean     Multifinder_Present;
extern  MenuHandle  Apple_Menu;
extern  MenuHandle  File_Menu;
extern  MenuHandle  Edit_Menu;
extern  Rect        Drag_Rect;
extern  Point       Screen_Center;
extern  short       Old_Button_Num;
extern  Boolean     Display_Hardware_Flag;
```

```
extern  Boolean     Color_QD_Present;
extern  short       Min_Pixel_Depth;
extern  Ptr         Info_Dialog_Storage;
extern  Ptr         Display_Window_Storage;
extern  WindowPtr   IV_Window_Ptr;
extern  DialogPtr   IV_Dialog_Ptr;
extern  Boolean     Display_Hardware_Flag;


/*++++++++++++++++++  Initialize the Toolbox  ++++++++++++++++++*/

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( NIL );
   FlushEvents( everyEvent, REMOVE_EVENTS );
   InitCursor();
}


/*++++++++  Verify that this program can run on this Mac  ++++++*/

void  Check_System( void )
{
   Boolean    gestalt_present;
   SysEnvRec  mac_info;

   SysEnvirons( curSysEnvVers, &mac_info );

   if ( mac_info.machineType < envMacII)
      Post_Error_Message(ERR_ROM_TOO_OLD, DO_TERMINATE_ERROR);

   if ( mac_info.systemVersion < 0x0604 )
      Post_Error_Message(ERR_SYSTEM_TOO_OLD, DO_TERMINATE_ERROR);

   gestalt_present = ( NGetTrapAddress(_Gestalt, OSTrap ) !=
                       NGetTrapAddress(_Unimplemented, OSTrap ) );
   if ( gestalt_present == FALSE )
      Post_Error_Message(ERR_NO_GESTALT, DO_TERMINATE_ERROR);
}
```

```
/*+++++++  Reserve low heap memory for window and dialog  ++++++*/

void  Reserve_Window_Memory( void )
{
   Info_Dialog_Storage    = NewPtr( sizeof ( DialogRecord ) );
   Display_Window_Storage = NewPtr( sizeof ( WindowRecord ) );
}


/*++++++++++++++  Initialize some program variables  ++++++++++++*/

void  Initialize_Variables( void )
{
   OSErr      err;
   long       response;

   Multifinder_Present = ( NGetTrapAddress(_WaitNextEvent, ToolTrap ) !=
                           NGetTrapAddress(_Unimplemented, ToolTrap ) );

   err = Gestalt( gestaltQuickdrawVersion, &response );

   if ( err == noErr )
   {
      if ( response == gestaltOriginalQD )
         Color_QD_Present = FALSE;
      else
         Color_QD_Present = TRUE;
   }
   else
      Post_Error_Message(ERR_CALL_TO_GESTALT_FAIL, DO_TERMINATE_ERROR);

   Set_Window_Drag_Boundaries();
   Set_Screen_Center();

   Old_Button_Num = HARDWARE_DITL_ITEM;
   Display_Hardware_Flag = TRUE;
}


/*++++++++++++  Initialize window drag boundaries  ++++++++++++*/

void  Set_Window_Drag_Boundaries( void )
{
   Drag_Rect = ( **( GrayRgn ) ).rgnBBox;
   Drag_Rect.left   += DRAG_EDGE;
```

```
   Drag_Rect.right  -= DRAG_EDGE;
   Drag_Rect.bottom -= DRAG_EDGE;
}


/*+++++  Determine center of monitor that has the menu bar  ++++*/

void  Set_Screen_Center( void )
{
   GDHandle  gd_handle;
   Rect      bnds_rect;

   gd_handle = GetMainDevice();

   bnds_rect = ( **( gd_handle ) ).gdRect;

   Screen_Center.h = ( bnds_rect.right /2 );
   Screen_Center.v = ( bnds_rect.bottom/2 ) + ( MENU_BAR_HEIGHT/2 );
}


/*++++++++++++++++++  Initialize the menu bar  ++++++*/

void  Set_Up_Menu_Bar( void )
{
   Handle  menu_bar_handle;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
   if ( menu_bar_handle == NIL )
      Post_Error_Message(ERR_MENU_BAR_FAIL, DO_TERMINATE_ERROR);

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   Apple_Menu = GetMHandle( APPLE_MENU_ID );
   File_Menu  = GetMHandle( FILE_MENU_ID );
   Edit_Menu  = GetMHandle( EDIT_MENU_ID );

   DisableItem( Edit_Menu, ENTIRE_MENU );

   AddResMenu( Apple_Menu, 'DRVR' );

   DrawMenuBar();
}
```

```
/*++++++++  Open the results window, but don't show it  ++++++++*/

void  Open_InnerView_Window( void )
{
   Str255  the_str;
   short   left, top;

   if ( Color_QD_Present && Min_Pixel_Depth > PIXEL_DEPTH_BW )
      IV_Window_Ptr = GetNewCWindow(IV_WIND_ID, Display_Window_Storage, IN_FRONT);
   else
      IV_Window_Ptr = GetNewWindow(IV_WIND_ID, Display_Window_Storage, IN_FRONT);

   if ( IV_Window_Ptr == NIL )
      Post_Error_Message(ERR_WIND_MEM_ALLOCATE_FAIL, DO_TERMINATE_ERROR);

   left = Screen_Center.h - ( WIND_WIDTH /2 );
   top  = Screen_Center.v - ( WIND_HEIGHT/2 );
   MoveWindow( IV_Window_Ptr, left, top, TRUE );
}


/*+++++++++  Open the info dialog, but don't show it  +++++++++*/

void  Open_InnerView_Dialog( void )
{
   short      left, top;
   Str255     the_str;

   IV_Dialog_Ptr = GetNewDialog(IV_DLOG_ID, Info_Dialog_Storage, IN_FRONT);

   if ( IV_Dialog_Ptr == NIL )
      Post_Error_Message(ERR_DIALOG_MEM_ALLOCATE_FAIL, DO_TERMINATE_ERROR);

   GetIndString( the_str, WIND_TITLE_STR_LIST, DLOG_INFO_TITLE );
   SetWTitle( IV_Dialog_Ptr, the_str );

   left = Screen_Center.h - ( DLOG_WIDTH /2 );
   top  = Screen_Center.v - ( DLOG_HEIGHT/2 );

   MoveWindow( IV_Dialog_Ptr, left, top, TRUE );

   Set_Radio_Buttons( IV_Dialog_Ptr, HARDWARE_DITL_ITEM );
   Display_Hardware_Flag = TRUE;
}
```

Utilities.c

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  Utilities.c                                            */
/*   Purpose:  Utility routines that may be used by more than          */
/*             one function, and by functions in more than one         */
/*             segment. An example would be Set_Radio_Button(),        */
/*             which is a generic function that can be used for        */
/*             any dialog box.                                         */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/


/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include "Defines.h"
#include "Utilities.h"


/*++  Make file aware of externally-declared global variables ++*/

extern  short  Old_Button_Num;


/*++++++  Display an alert with descriptive error message  +++++*/

void  Post_Error_Message( short error_num, Boolean terminate )
{
   Str255  the_str;

   GetIndString( the_str, ERR_STR_LIST, error_num );
   ParamText( the_str, "\p", "\p", "\p" );
   StopAlert( ERR_ALRT_ID, NIL );

   if ( terminate == DO_TERMINATE_ERROR )
      ExitToShell();
}


/*+++++++++++++++  Respond to click in a check box  +++++++++++++*/

void Set_Check_Box( DialogPtr the_dialog, short the_item )
{
   Handle  item_handle;
```

```
    short   item_type;
    Rect    item_rect;
    int     old_value;

    GetDItem( the_dialog, the_item, &item_type, &item_handle, &item_rect );
    old_value = GetCtlValue( ( ControlHandle )item_handle );

    if ( old_value == CONTROL_ON )
        SetCtlValue( (ControlHandle )item_handle, CONTROL_OFF );
    else
        SetCtlValue( (ControlHandle )item_handle, CONTROL_ON );
}


/*++++++++++++  Respond to click in a radio button  ++++++++++++*/

void  Set_Radio_Buttons( DialogPtr the_dialog, short new_button_num )
{
    Handle  item_handle;
    short   item_type;
    Rect    item_rect;

    GetDItem(the_dialog, Old_Button_Num, &item_type, &item_handle, &item_rect);
    SetCtlValue( ( ControlHandle )item_handle, CONTROL_OFF );

    GetDItem (the_dialog, new_button_num, &item_type, &item_handle, &item_rect);
    SetCtlValue( ( ControlHandle )item_handle, CONTROL_ON );

    Old_Button_Num = new_button_num;
}


/*++++++++  Get user-entered text from a text edit item  +++++++*/

void  Get_Text_From_Edit( DialogPtr the_dialog, short edit_item,
                          Str255 the_string )
{
    Handle  item_handle;
    short   item_type;
    Rect    item_rect;

    GetDItem( the_dialog, edit_item, &item_type, &item_handle, &item_rect );
    GetIText( item_handle, the_string );
}
```

InnerViewII.c

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   File    :  InnerViewII.c                                              */
/*   Purpose:  Bulk of the InnerViewII program. Use a modeless             */
/*             dialog that allows the user to choice between               */
/*             viewing Mac hardware or software features, then             */
/*             display the results in a window.                            */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/


/*++++++++++++++++++++++   Include Files   ++++++++++++++++++++++*/

#include "Defines.h"
#include "Globals.h"
#include "Initialize.h"
#include "Utilities.h"
#include <GestaltEqu.h>


/*++++++++++++++++++++   Function prototypes   ++++++++++++++++++*/

short    Get_Min_Pixel_Depth( void );
short    Get_Pixel_Depth( GDHandle );
void     Handle_One_Event( void );
Boolean  Handle_Dialog_Event( void );
void     Get_Dialog_Info( DialogPtr the_dialog );
void     Enable_Disable_Menu_Items( void );
void     Handle_Keystroke( void );
void     Handle_Mouse_Down( void );
void     Handle_Menu_Choice( long );
void     Handle_Apple_Choice( short );
void     Handle_File_Choice( short );
void     Handle_Edit_Choice( short );
void     Handle_Update( void );
void     Update_IV_Window( void );
void     Get_Hardware_Information( void );
void     Display_Hardware_Information( void );
void     Get_Software_Information( void );
void     Display_Software_Information( void );
void     Draw_Owner_Information( void );
void     Draw_Mac_Picture( void );
void     Draw_System_Info_Headings( void );
```

```
/*++++++++++++++++++++++  main listing  ++++++++++++++++++++++++*/

void main( void )
{
   MaxApplZone();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

   Initialize_Toolbox();
   Check_System();
   Reserve_Window_Memory();
   Initialize_Variables();
   Set_Up_Menu_Bar();
   Min_Pixel_Depth = Get_Min_Pixel_Depth();

   Open_InnerView_Window();
   Open_InnerView_Dialog();

   UnloadSeg( Initialize_Toolbox );

   while ( All_Done == FALSE )
   {
      Handle_One_Event();
      UnloadSeg( Set_Check_Box );
   }
}


/*+++++++++  Find pixel depth of lowest color monitor  +++++++++*/

short  Get_Min_Pixel_Depth( void )
{
   GDHandle  current_device;
   short     pixel_depth;
   short     min_depth;

   min_depth = PIXEL_DEPTH_MAX_COLOR;
   current_device = GetDeviceList();
   while ( current_device != NIL )
   {
      pixel_depth = Get_Pixel_Depth( current_device );
      if ( pixel_depth < min_depth )
         min_depth = pixel_depth;
```

```
      current_device = GetNextDevice( current_device );
   }
   return  min_depth;
}


/*+++++++++++++++  Get pixel depth of one monitor  +++++++++++++++*/

short  Get_Pixel_Depth( GDHandle the_device )
{
   PixMapHandle  screenPMapH;
   short         pixel_depth;

   screenPMapH = ( **the_device ).gdPMap;
   pixel_depth = ( **screenPMapH ).pixelSize;
   return  pixel_depth ;
}


/*++++++++++++++++++++  Handle a single event  ++++++++++++++++++*/

void  Handle_One_Event( void )
{
   Boolean    event_was_dialog = FALSE;

   if ( Multifinder_Present == TRUE )
      WaitNextEvent( everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION );
   else
   {
      SystemTask();
      GetNextEvent( everyEvent, &The_Event );
   }

   event_was_dialog = Handle_Dialog_Event();

   if ( event_was_dialog == FALSE )
   {
      switch ( The_Event.what )
      {
         case activateEvt:
            Enable_Disable_Menu_Items();

         case keyDown:
            Handle_Keystroke();
            break;
```

```
            case mouseDown:
               Handle_Mouse_Down();
               break;

            case updateEvt:
               if ( Color_QD_Present == TRUE )
                  Min_Pixel_Depth = Get_Min_Pixel_Depth();
               Handle_Update();
               break;
         }
      }
   }


/*+++++++++++++++  Handle a dialog-related event  +++++++++++++++*/

Boolean  Handle_Dialog_Event( void )
{
   Boolean    event_was_dlog = FALSE;
   DialogPtr  the_dialog;
   short      the_item;
   short      alert_item;
   Str255     the_str;

   if ( FrontWindow() != NIL )
   {
      if ( IsDialogEvent( &The_Event )  )
      {
         if ( DialogSelect( &The_Event, &the_dialog, &the_item ) )
         {
            switch ( the_item )
            {
               case OK_BUTTON_DITL_ITEM:
                  Get_Dialog_Info( the_dialog );
                  HideWindow( IV_WindoXw_Ptr );
                  if ( Display_Hardware_Flag == TRUE )
                     GetIndString(the_str, WIND_TITLE_STR_LIST, WIND_HARDWARE_TITLE);
                  else
                     GetIndString(the_str, WIND_TITLE_STR_LIST, WIND_SOFTWARE_TITLE);
                  SetWTitle( IV_Window_Ptr, the_str );
                  ShowWindow( IV_Window_Ptr );
                  SelectWindow( IV_Window_Ptr );
                  break;
```

```
                case CHECK_DITL_ITEM:
                   Set_Check_Box( the_dialog, the_item );
                   break;

                case HARDWARE_DITL_ITEM:
                   Set_Radio_Buttons( the_dialog, the_item );
                   Display_Hardware_Flag = TRUE;
                   break;

                case SOFTWARE_DITL_ITEM:
                   Set_Radio_Buttons( the_dialog, the_item );
                   Display_Hardware_Flag = FALSE;
                   break;
             }
             event_was_dlog = TRUE;
          }
       }
   }
   return ( event_was_dlog );
}


/*+++++  Adjust Edit menu in response to activate event  +++++++*/

void  Enable_Disable_Menu_Items( void )
{
   if ( FrontWindow() == IV_Dialog_Ptr )
      EnableItem( Edit_Menu, ENTIRE_MENU );
   else
      DisableItem( Edit_Menu, ENTIRE_MENU );

   DrawMenuBar();
}


/*++++++++++++++++++++  Handle a keystroke ++++++++++++++++++++*/

void  Handle_Keystroke( void )
{
   short  chr;
   long   menu_choice;

   chr = The_Event.message & charCodeMask;

   if ( ( The_Event.modifiers & cmdKey ) != 0 )
```

```
    {
        if ( The_Event.what != autoKey )
        {
            menu_choice = MenuKey( chr );
            Handle_Menu_Choice( menu_choice );
        }
    }
}


/*++++++++++++  Handle a click of the mouse button  ++++++++++++++*/

void  Handle_Mouse_Down( void )
{
    WindowPtr  the_window;
    short      the_part;
    long       menu_choice;

    the_part = FindWindow( The_Event.where, &the_window );

    switch ( the_part )
    {
        case inMenuBar:
            menu_choice = MenuSelect( The_Event.where );
            Handle_Menu_Choice( menu_choice );
            break;

        case inSysWindow:
            SystemClick( &The_Event, the_window);
            break;

        case inDrag:
            DragWindow( the_window, The_Event.where, &Drag_Rect );
            break;

        case inGoAway:
            if ( TrackGoAway( the_window, The_Event.where ) )
                HideWindow( the_window );
            break;

        case inContent:
            SelectWindow( the_window );
            break;
    }
}
```

```
/*+++++++++++++++++  Handle a click on a menu  ++++++++++++++++++*/

void  Handle_Menu_Choice( long menu_choice )
{
   int  the_menu;
   int  the_menu_item;

   if ( menu_choice != 0 )
   {
      the_menu = HiWord( menu_choice );
      the_menu_item = LoWord( menu_choice );

      switch ( the_menu )
      {
         case APPLE_MENU_ID :
            Handle_Apple_Choice( the_menu_item );
            break;

         case FILE_MENU_ID :
            Handle_File_Choice( the_menu_item );
            break;

         case EDIT_MENU_ID:
            Handle_Edit_Choice( the_menu_item );
            break;
      }
      HiliteMenu( 0 );
   }
}


/*++++++++++++  Handle a click of in the Apple menu  ++++++++++++*/

void  Handle_Apple_Choice( short the_item )
{
   Str255  desk_acc_name;
   int     desk_acc_number;

   switch ( the_item )
   {
      case ABOUT_ITEM :
         NoteAlert( ABOUT_ALRT_ID, NIL ):
         break;
```

```
      default :
         GetItem( Apple_Menu, the_item, desk_acc_name );
         desk_acc_number = OpenDeskAcc( desk_acc_name );
         break;
   }
}


/*++++++++++++++  Handle a click in the File menu  ++++++++++++++*/

void  Handle_File_Choice( short the_item )
{
   switch ( the_item )
   {
      case NEW_ITEM :
      case TO_FRONT_ITEM :
         ShowWindow( IV_Dialog_Ptr );
         SelectWindow( IV_Dialog_Ptr );
         break;

      case QUIT_ITEM :
         All_Done = TRUE;
         break;
   }
}


/*++++++++++++++  Handle a click in the Edit menu  ++++++++++++++*/

void  Handle_Edit_Choice( short the_item )
{
   switch ( the_item )
   {
      case CUT_ITEM :
         DlgCut ( IV_Dialog_Ptr );
         break;

      case COPY_ITEM :
         DlgCopy ( IV_Dialog_Ptr );
         break;

      case PASTE_ITEM :
         DlgPaste ( IV_Dialog_Ptr );
         break;
```

```
    }
}


/*+++++++++++++++++++  Handle an update event  +++++++++++++++++*/

void  Handle_Update( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;

   if ( the_window == IV_Window_Ptr )
      Update_IV_Window();
}


/*+++++++++++++++  Update the InnerView window  +++++++++++++++++*/

void  Update_IV_Window( void )
{
   GrafPtr    old_port;
   WindowPtr  the_window;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   TextFont(systemFont);
   TextSize(12);

   BeginUpdate( IV_Window_Ptr );

      Draw_Mac_Picture();
      Draw_Owner_Information();
      Draw_System_Info_Headings();

      if ( Display_Hardware_Flag == TRUE )
      {
         Get_Hardware_Information();
         Display_Hardware_Information();
      }
      else
      {
         Get_Software_Information();
         Display_Software_Information();
```

```
    }

  EndUpdate( IV_Window_Ptr );

  SetPort( old_port );
}


/*++++++++++++  Get software info about user's machine  +++++++++*/

void  Get_Hardware_Information( void )
{
  OSErr    err;
  long     response;
  Boolean  failed = FALSE;

  err = Gestalt( gestaltProcessorType, &response );
  if ( err == noErr )
    Mac_CPU = response;
  else
  {
    Mac_CPU = GESTALT_ERR_TYPE;
    failed = TRUE;
  }

  err = Gestalt( gestaltFPUType, &response );
  if ( err == noErr )
    Mac_FPU = response;
  else
  {
    Mac_FPU = GESTALT_ERR_TYPE;
    failed = TRUE;
  }

  err = Gestalt( gestaltPhysicalRAMSize, &response );
  if ( err == noErr )
  {
    Mac_RAM_Str_Error = FALSE;
    NumToString( response, Mac_RAM_Str );
  }
  else
  {
    Mac_RAM_Str_Error = TRUE;
    failed = TRUE;
  }
```

```
    if ( failed == TRUE )
        Post_Error_Message(ERR_CALL_TO_GESTALT_FAIL, DONT_TERMINATE_ERROR);
}


/*++++++++  Display hardware info about user's machine  ++++++++*/

void  Display_Hardware_Information( void )
{
    Str255   the_str;
    short    the_list;
    short    the_index;

    switch ( Mac_CPU )
    {
        case gestalt68000:
            the_index = 1;
            break;
        case gestalt68010:
            the_index = 2;
            break;
        case gestalt68020:
            the_index = 3;
            break;
        case gestalt68030:
            the_index = 4;
            break;
        case gestalt68040:
            the_index = 5;
            break;
        default:
            the_index = 6;
            break;
    }
    the_list = CPU_TYPE_STR_LIST;
    GetIndString( the_str, the_list, the_index );
    MoveTo( RESULT_X, HEADING_Y + LINE_HEIGHT );
    DrawString( the_str );

    switch ( Mac_FPU )
    {
        case gestaltNoFPU:
            the_index = 1;
            break;
```

```
      case gestalt68881:
         the_index = 2;
         break;
      case gestalt68882:
         the_index = 3;
         break;
      case gestalt68040FPU:
         the_index = 4;
         break;
      default:
         the_index = 5;
         break;
   }
   the_list = FPU_TYPE_STR_LIST;
   GetIndString( the_str, the_list, the_index );
   MoveTo( RESULT_X, HEADING_Y + ( 2 * LINE_HEIGHT ) );
   DrawString( the_str );

   if ( Mac_RAM_Str_Error == TRUE )
      GetIndString( Mac_RAM_Str, RAM_STR_LIST, 1 );
   MoveTo( RESULT_X, HEADING_Y + ( 3 * LINE_HEIGHT ) );
   DrawString( Mac_RAM_Str );
}


/*++++++++++++  Get software info about user's machine  +++++++++*/

void  Get_Software_Information( void )
{
   OSErr  err;
   long   response;
   short  digit;
   long   temp;
   short  i;
   Boolean  failed = FALSE;

   err = Gestalt( gestaltSystemVersion, &response );
   if ( err == noErr )
   {
      Mac_RAM_Str_Error = FALSE;
      for ( i=1; i <= 3; i++ )
      {
         temp = response;
         if ( i == 1 )
            digit = ( temp &= 0x0F00 ) / 0x0100;
```

```
            else if ( i == 2 )
               digit = ( temp &= 0x00F0 ) / 0x0010;
            else
               digit = ( temp &= 0x000F ) / 0x0001;
            digit += ASCII_ZERO;
            Mac_Sys_Str[i] = digit;
         }
         Mac_Sys_Str[0] = 3;
      }
      else
      {
         Mac_RAM_Str_Error = TRUE;
         failed = TRUE;
      }

      err = Gestalt( gestaltQuickdrawVersion, &response );
      if ( err == noErr )
      {
         Mac_Has_Color_QD = response;
         Mac_QD_Version = response;
      }
      else
      {
         Mac_Has_Color_QD = GESTALT_ERR_TYPE;
         Mac_QD_Version = GESTALT_ERR_TYPE;
         failed = TRUE;
      }

      if ( failed == TRUE )
         Post_Error_Message(ERR_CALL_TO_GESTALT_FAIL, DONT_TERMINATE_ERROR);
}


/*++++++++  Display software info about user's machine  ++++++++*/

void  Display_Software_Information( void )
{
   Str255   the_str;
   short    the_list;
   short    the_index;

   if ( Mac_Sys_Str_Error == TRUE )
      GetIndString( Mac_Sys_Str, SYSTEM_STR_LIST, 1 );
   MoveTo( RESULT_X, HEADING_Y + LINE_HEIGHT );
   DrawString( Mac_Sys_Str );
```

```
    if ( Mac_Has_Color_QD == GESTALT_ERR_TYPE )
       the_index = 3;
    else
    {
       if ( Mac_Has_Color_QD == gestaltOriginalQD )
          the_index = 1;
       else
          the_index = 2;
    }
    the_list = COLOR_QUICKDRAW_STR_LIST;
    GetIndString( the_str, the_list, the_index );
    MoveTo( RESULT_X, HEADING_Y + ( 2 * LINE_HEIGHT ) );
    DrawString( the_str );

    switch ( Mac_QD_Version )
    {
       case gestaltOriginalQD:
          the_index = 1;
          break;
       case gestalt8BitQD:
          the_index = 2;
          break;
       case gestalt32BitQD:
          the_index = 3;
          break;
       case gestalt32BitQD11:
          the_index = 4;
          break;
       case gestalt32BitQD12:
          the_index = 5;
          break;
       case gestalt32BitQD13:
          the_index = 6;
          break;
       default:
          the_index = 7;
          break;
    }
    the_list = QUICKDRAW_VER_STR_LIST;
    GetIndString( the_str, the_list, the_index );
    MoveTo( RESULT_X, HEADING_Y + ( 3 * LINE_HEIGHT ) );
    DrawString( the_str );
}
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void  Draw_Owner_Information( void )
{
   Str255  the_str;

   GetIndString( the_str, OWNER_INFO_STR_LIST, OWNER_TITLE );
   MoveTo( HEADING_X, OWNER_Y );
   DrawString( the_str );

   MoveTo( RESULT_X, OWNER_Y );
   if ( Print_Name == TRUE )
      DrawString( Name_Str );
   else
   {
      GetIndString( the_str, OWNER_INFO_STR_LIST, OWNER_NOT_AVAILABLE );
      DrawString( the_str );
   }
}


/*++++++++++++++  Draw a PICT to the results window  ++++++++++++*/

void  Draw_Mac_Picture( void )
{
   GrafPtr    old_port;
   PicHandle  the_pict;
   Rect           pict_rect;
   short      pict_wd, pict_ht;
   short      pict_id;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   if ( Min_Pixel_Depth > PIXEL_DEPTH_BW )
      pict_id = MAC_PICT_COLOR_ID;
   else
      pict_id = MAC_PICT_BW_ID;

   the_pict = GetPicture( pict_id );
   if ( the_pict == NIL )
      Post_Error_Message(ERR_PICT_MEM_ALLOCATE_FAIL, DONT_TERMINATE_ERROR);

   pict_rect = ( **( the_pict ) ).picFrame;
```

```
    pict_wd  = pict_rect.right - pict_rect.left;
    pict_ht = pict_rect.bottom - pict_rect.top;
    SetRect ( &pict_rect, PICT_L, PICT_T, PICT_L + pict_wd, PICT_T + pict_ht );

    DrawPicture( the_pict, &pict_rect );

    SetPort( old_port );
}


/*++++++++++++  Draw headings to the results window  ++++++++++++*/

void  Draw_System_Info_Headings( void )
{
    short   index;
    short   headings;
    short   str_list_id;
    Str255  the_str;

    if ( Display_Hardware_Flag == TRUE )
    {
        headings = NUM_HARDWARE_HEADINGS;
        str_list_id = HARDWARE_TITLES_STR_LIST;
    }
    else
    {
        headings = NUM_SOFTWARE_HEADINGS;
        str_list_id = SOFTWARE_TITLES_STR_LIST;
    }

    for ( index = 1; index <= headings; index++ )
    {
        GetIndString(the_str, str_list_id, index );
        MoveTo( HEADING_X, HEADING_Y + ( index * LINE_HEIGHT ) );
        DrawString( the_str );
    }
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void  Get_Dialog_Info( DialogPtr the_dialog )
```

```
{
   Handle  item_handle;
   short   item_type;
   Rect    item_rect;
   int     cntl_value;

   Get_Text_From_Edit( the_dialog, NAME_DITL_ITEM, Name_Str );

   GetDItem( the_dialog, CHECK_DITL_ITEM, &item_type, &item_handle, &item_rect );
   cntl_value = GetCtlValue( ( ControlHandle )item_handle );

   if ( cntl_value == CONTROL_ON )
      Print_Name = TRUE;
   else
      Print_Name = FALSE;
}
```

Stepping through the header files

The *InnerViewII* program is the culmination of nine chapters, so you've seen much of the code before. Let's spend most of the next several pages on things that were covered in this chapter, and things done a little differently than in the past.

Defines.h

From the preceding eight chapters you have a pretty good idea of what many of the *#defines* represent. I'll take the liberty of summarizing things here.

The first two dozen #define statements all involve 'STR#' resources. Each of the eleven 'STR#' lists has a separate *#define.* I ended the name of each with *"_STR_LIST"* for consistency. I created constants for some of the string entries in a few of the lists.

IV_DLOG_ID is the ID of the 'DLOG' resource for the information dialog box. Each of the five enabled items in the dialog each has its own #define. To center the dialog box, you supply the dialog width and height to have the contents—*DLOG_WIDTH* and *DLOG_HEIGHT.*

The results display window has a 'WIND' ID of *IV_WIND_ID*. The display window, like the dialog box, has a #define for width and height—*WIND_WIDTH* and *WIND_HEIGHT*.

The About InnerViewII 'ALRT' has an ID of *ABOUT_ALRT_ID*. The 'ALRT' used to display error messages has an ID of *ERR_ALRT_ID*.

There are 11 *#defines* for the menu bar. The menu bar's 'MBAR' and each of the three 'MENU' resources have their own constants. Each item within a menu has a *#define* for the constant.

The two 'PICT' resources have *#defines—MAC_PICT_BW_ID* for the monochrome picture, *MAC_PICT_COLOR_ID* for the color version.

The host of "standard" *#defines* (those I've been using in every program) are present here. I've added two constants to signal the error alert so that it knows whether or not to terminate the program. These constants are *DO_TERMINATE_ERROR* and *DONT_TERMINATE_ERROR*.

There are nine *#defines* for positioning text in the display window. I like to keep these pixel values in just one spot so that if I want to rearrange things the work will be minimal. *HEADING_X* and *HEADING_Y* are examples of this type of constant.

Globals.h

InnerViewII keeps all of the global variables in *Globals.h*. Like the *#defines*, you've seen many of these before. Here's a look at the new globals.

Both the dialog box and the window have memory reserve set aside for them early in the program. *Info_Dialog_Storage* and *Display_Window_Storage* are the pointers to these areas.

The results of the six machine features that InnerViewII checks for are kept in a global variable. Each begins with *"Mac_"*.

Initialize.h *and* Utilities.h

Initialize.h makes the functions in *Initialize.c* that are known to other files. Any file that uses one or more functions found in *Initialize.c* will use *Initialize.h* in a *#include* directive. The same applies to functions found in *Utilities.c*; they're made public by *Utilities.h*

Stepping through initialize.c

Each routine in *Initialize.c* is a one-time-only routine. Once called, it will never be called again. You'll see the significance of removing these routines from the main source file, *InnerViewII.c.* as you step through that file.

Checking the system

Check_System() was developed in Chapter 8. Always make a few standard checks to verify that your program is capable of running on the machine that started it. InnerViewII relies heavily on information obtained through calls to *Gestalt()*. So you have to make sure the Mac has the Gestalt trap.

If any of the checks fail, *Check_System()* calls the program's error handling routine *Post_Error_Message()*.

```
void  Check_System( void )
{
   Boolean    gestalt_present;
   SysEnvRec  mac_info;

   SysEnvirons( curSysEnvVers, &mac_info );

   if ( mac_info.machineType < envMacII)
      Post_Error_Message(ERR_ROM_TOO_OLD, DO_TERMINATE_ERROR);

   if ( mac_info.systemVersion < 0x0604 )
      Post_Error_Message(ERR_SYSTEM_TOO_OLD, DO_TERMINATE_ERROR);

   gestalt_present = ( NGetTrapAddress(_Gestalt, OSTrap ) !=
                       NGetTrapAddress(_Unimplemented, OSTrap ) );
   if ( gestalt_present == FALSE )
```

```
        Post_Error_Message(ERR_NO_GESTALT, DO_TERMINATE_ERROR);
}
```

Reserving memory

This chapter suggested that you reserve memory for windows. You can do the same for dialogs. *Reserve_Memory()* does just that. When you get to *InnerViewII.c* you'll see that this function is called early so that the memory that is reserved low in the heap. This avoids fragmentation.

```
void  Reserve_Window_Memory( void )
{
   Info_Dialog_Storage    = NewPtr( sizeof ( DialogRecord ) );
   Display_Window_Storage = NewPtr( sizeof ( WindowRecord ) );
}
```

Initializing variables

The routine *Initialize_Variables()* gives some program globals their initial values, based on the Macintosh *InnerViewII* is running on. *Old_Button_Num* keeps track of which of two radio buttons is set in the information dialog box. When the dialog box is opened you'll want one of the two buttons to go on. Since you're telling the Mac to turn on the hardware button, set the global flag to signal that this is the case.

```
void  Initialize_Variables( void )
{
   OSErr      err;
   long       response;

   Multifinder_Present = ( NGetTrapAddress(_WaitNextEvent, ToolTrap ) !=
                           NGetTrapAddress(_Unimplemented, ToolTrap ) );

   err = Gestalt( gestaltQuickdrawVersion, &response );

   if ( err == noErr )
   {
      if ( response == gestaltOriginalQD )
         Color_QD_Present = FALSE;
      else
         Color_QD_Present = TRUE;
   }
```

```
    Post_Error_Message(ERR_CALL_TO_GESTALT_FAIL, DO_TERMINATE_ERROR);

    Set_Window_Drag_Boundaries();
    Set_Screen_Center();

    Old_Button_Num = HARDWARE_DITL_ITEM;
    Display_Hardware_Flag = TRUE;
}
```

Opening the window and dialog box

The calls to open the window and dialog (*GetNewWindow()* and *GetNewDialog()*) pass the global storage pointers set earlier in the program. The *WindowRecord* and *DialogRecord* that hold the information for each are now safely tucked in low memory.

InnerViewII opens the window and dialog immediately, but it doesn't show them. Both the 'DLOG' and 'WIND' resources are marked as invisible. A call to *ShowWindow()* to make either visible. Why open them without displaying them? Here's the story.

InnerViewII manages its window and dialog boxes in a manner different from those managed by other programs you've seen here. *InnerViewII* opens both and leaves them open for the duration of the program. To display either one the program uses *ShowWindow()*. To close one, it uses *HideWindow()*. Unlike most programs that use the Toolbox routine *CloseWindow()*, *InnerViewII* never really gets rid of either the window or dialog; it simply appears that way to the user. When the user selects the New command you need only call *ShowWindow()* to make the dialog box visible again: Please don't call *GetNewDialog()*.

This programs window-handling technique enables only one dialog box and one window. If the program's New command opened a new window each time, (allowing multiple windows on the screen) *InnerViewII* would quickly lose track of which window was which. With *InnerViewII*, you can safely assume that any event that involves an application window also involves the display results window. I keep a global pointer to it and use that for the duration of the program. The same applies to the one and only dialog box in the program.

```
void  Open_InnerView_Window( void )
{
```

```
    Str255  the_str;
    short   left, top;

    if ( Color_QD_Present && Min_Pixel_Depth > PIXEL_DEPTH_BW )
       IV_Window_Ptr = GetNewCWindow(IV_WIND_ID, Display_Window_Storage, IN_FRONT);
    else
       IV_Window_Ptr = GetNewWindow(IV_WIND_ID, Display_Window_Storage, IN_FRONT);

    if ( IV_Window_Ptr == NIL )
       Post_Error_Message(ERR_WIND_MEM_ALLOCATE_FAIL, DO_TERMINATE_ERROR);

    left = Screen_Center.h - ( WIND_WIDTH /2 );
    top  = Screen_Center.v - ( WIND_HEIGHT/2 );
    MoveWindow( IV_Window_Ptr, left, top, TRUE );
}


void  Open_InnerView_Dialog( void )
{
    short      left, top;
    Str255     the_str;

    IV_Dialog_Ptr = GetNewDialog(IV_DLOG_ID, Info_Dialog_Storage, IN_FRONT);

    if ( IV_Dialog_Ptr == NIL )
       Post_Error_Message(ERR_DIALOG_MEM_ALLOCATE_FAIL, DO_TERMINATE_ERROR);

    GetIndString( the_str, WIND_TITLE_STR_LIST, DLOG_INFO_TITLE );
    SetWTitle( IV_Dialog_Ptr, the_str );

    left = Screen_Center.h - ( DLOG_WIDTH /2 );
    top  = Screen_Center.v - ( DLOG_HEIGHT/2 );

    MoveWindow( IV_Dialog_Ptr, left, top, TRUE );

    Set_Radio_Buttons( IV_Dialog_Ptr, HARDWARE_DITL_ITEM );
    Display_Hardware_Flag = TRUE;
}
```

By the way, don't be alarmed that one of the dialog box radio buttons is set with a call to *Set_Radio_Button()*, even though the dialog isn't visible. The fact that the dialog is invisible doesn't prevent us from changing item settings. When the dialog becomes visible the proper radio button will be set.

Stepping through *Utilities.c*

The program calls routines that can be called by several functions or that can be copied and pasted into other programs' "utility functions." They're grouped together in one convenient place—*Utilities.c*. This makes it especially handy for use in other programs. You don't have to bother copying individual routines; you just copy the entire file!

You've used two of the four routines in other programs. *Set_Check_Box()* and *Set_Radio_Button()* control items in a dialog box. The *Get_Text_From_Edit()* function is just a packaging of material covered in Chapter 6. It gets the user-entered text from an edit text item.

The forth routine is *Post_Error_Message()*. This routine was covered in great detail in this chapter.

Stepping through *InnerViewII.c*

There's bad news and good news with *InnerViewII.c*. The bad news is that *InnerViewII.c* contains 21 functions. The good news is that only a handful contain new information. This discussion covers only those routines that do.

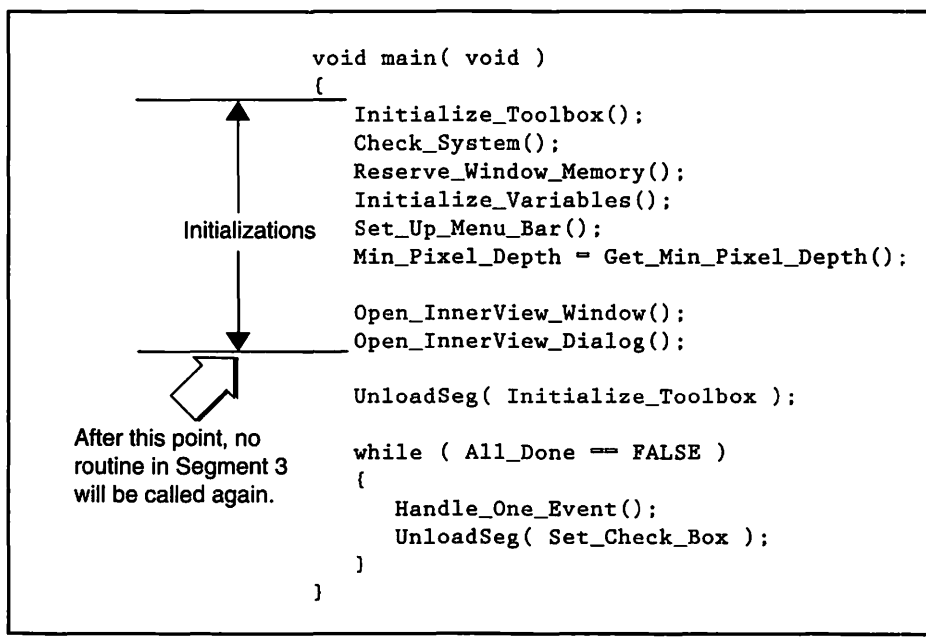
The main() function

The *main()* function starts out with standard initializations. After that, it opens, but doesn't display, both the dialog box and the window. That will come later on.

InnerViewII uses segmentation. Although InnerViewII doesn't need segmentation because of its small size, I segmented it to try out some of the techniques discussed in this chapter.

When I discussed the header files I said that there was special significance in taking all of the initialization routines out of *InnerViewII.c*. I was hinting that it would help you in your use of segmentation. I've placed all of the one-time-only routines together in one segment—Segment 3, and I've made sure that any routine that is called more than once does not appear in Segment 3. After all of the initialization routines have been called you can unload the segment and know that it will never be loaded again. That means that the memory the routines would normally occupy will be free permanently.

Once *UnloadSeg()* is called in Segment 3, the Memory Manager will unload it if memory becomes tight. If even one of the routines in Segment 3 is called more than once, Segment 3 will have to be reloaded at a later time. Since none of the functions will be called again, the segment is out of memory for good. Figure 9–51 illustrates this.




```
void main( void )
{
    Initialize_Toolbox();
    Check_System();
    Reserve_Window_Memory();
    Initialize_Variables();
    Set_Up_Menu_Bar();
    Min_Pixel_Depth = Get_Min_Pixel_Depth();

    Open_InnerView_Window();
    Open_InnerView_Dialog();

    UnloadSeg( Initialize_Toolbox );

    while ( All_Done == FALSE )
    {
       Handle_One_Event();
       UnloadSeg( Set_Check_Box );
    }
}
```

Figure 9–51. *Initialization routines can be unloaded permanently*

When I covered segmentation, I suggested that you place all calls to *UnloadSeg()* at the bottom of your main event loop. The reason I call *UnloadSeg()* repeatedly on each segment is because segments may be shuffling in and out of memory throughout the course of a program's life. I don't know what's in and what's out, so I play it safe. Segment 3 is the exception to this. I know that when I'm done with it I'll never use it again. If it gets unloaded, it will stay unloaded. So a single call to *UnloadSeg()* does the trick.

The single call to *UnloadSeg()* in the event loop looks a little lonely. A larger application might have 10, 20, or more segments. To keep things looking clean you might do something like this:

```
while ( All_Done == FALSE )
{
   Handle_One_Event();
   Unload_All_Segments();
}

void  Unload_All_Segments( void )
{
   UnloadSeg( Seg_A_Routine_Name );
   UnloadSeg( Seg_B_Routine_Name );
   UnloadSeg( Seg_C_Routine_Name );
   [ unload each segment here ]
}
```

Right now you might be saying, "What is the point of all this?" I understand. The entire *InnerViewII* program is only 11K of code; it's extremely unlikely that any of the segments will have to be unloaded! Additionally, Segment 3 is less than 800 bytes of code—not exactly a whopping savings of RAM, even if it were to be unloaded. But remember, this is technique. I want to demonstrate how it's done. This same technique applies to a program that is 500 K in size, with 25 K of initialization code. And truthfully now, would you really want to step through a more practical example like that?

```
void main( void )
{
   MaxApplZone();
   MoreMasters();
   MoreMasters();
   MoreMasters();
   MoreMasters();

   Initialize_Toolbox();
   Check_System();
   Reserve_Window_Memory();
   Initialize_Variables();
   Set_Up_Menu_Bar();
   Min_Pixel_Depth = Get_Min_Pixel_Depth();

   Open_InnerView_Window();
   Open_InnerView_Dialog();

   UnloadSeg( Initialize_Toolbox );

   while ( All_Done == FALSE )
   {
      Handle_One_Event();
      UnloadSeg( Set_Check_Box );
   }
}
```

Dimming a menu

InnerViewII allows use of the Edit menu commands in the edit text item of the modeless dialog box. Whenever that dialog is the frontmost window, the edit menu is active and available. But if the display results window is in the forefront, it wouldn't make much sense to have the Edit menu enabled. There is nothing to edit in the window. This is a realistic example of when to disable and enable a menu.

InnerViewII handles the situation by simply calling either *EnableItem()* or *DisableItem()* for the Edit menu at the appropriate time. And when is that? Whenever an activate event occurs. Any time a window or dialog is activated the program checks to see which one came to the forefront. If it's the dialog box, it enables the menu. If it's the window, it disables it. Here's a fragment of the code that handles things:

```
void  Handle_One_Event( void )
{
   [ get one event ]

   if ( event_was_dialog == FALSE )
   {
      switch ( The_Event.what )
      {
         case activateEvt:
            Enable_Disable_Menu_Items();

         [ handle other event types ]
      }
   }
}


void  Enable_Disable_Menu_Items( void )
{
   if ( FrontWindow() == IV_Dialog_Ptr )
      EnableItem( Edit_Menu, ENTIRE_MENU );
   else
      DisableItem( Edit_Menu, ENTIRE_MENU );

   DrawMenuBar();
}
```

Updating the window

InnerViewII accepts input from the user by way of a modeless dialog box. Modal dialogs restrict the user, so use modeless whenever possible. Once you know the technique for handling them—and you learned from Chapter 6, the few extra lines of code you need to write will be greatly appreciated by the user.

The user operates the dialog's radio buttons to signal which machine parameters he's interested in seeing. When the user clicks the OK button, the window that displays the results should be cleared. If software information was displayed and the user now wants to see hardware information, he doesn't want to see something drawn on top of the old information.

InnerViewII keeps a global *WindowPtr* variable that points to the window. After initially calling *GetNewWindow()*, the program never actually closes and opens the window, it just hides and shows it. You take advantage of this simple idea when you clear the window.

A click on the OK button, calls *HideWindow()*. You then set the window's title to the proper text and call *ShowWindow()* and *SelectWindow()*. The Toolbox routine *ShowWindow()* does just that; it shows a window. Once you've made the window visible, you call another Toolbox routine—*SelectWindow()*. This function brings the window to the front and generates an update event. And, as you recall from Chapter 5, an update event is the action that triggers the program to redraw a window. So by hiding the window and then showing it, you get the same effect as clearing it and drawing to it. This assumes that you have a *Handle_Update()* routine that does the drawing. And of course you do. But first, here's the part of *Handle_Dialog_Event()* that plays hide and seek with the window.

```
Boolean  Handle_Dialog_Event( void )
{
   [ check for dialog event and respond to it ]

      switch ( the_item )
      {
         case OK_BUTTON_DITL_ITEM:
            Get_Dialog_Info( the_dialog );
            HideWindow( IV_Window_Ptr );
            if ( Display_Hardware_Flag == TRUE )
               GetIndString(the_str, WIND_TITLE_STR_LIST, WIND_HARDWARE_TITLE);
            else
               GetIndString(the_str, WIND_TITLE_STR_LIST, WIND_SOFTWARE_TITLE);
            SetWTitle( IV_Window_Ptr, the_str );
            SelectWindow( IV_Window_Ptr );
            break;

         [ handle clicks on other items ]

}
```

After *SelectWindow()* generates an update event, *Handle_One_Event()* is called and in turn calls *Handle_Update()*. Since you arrive at this routine when either the dialog or the window comes to the forefront, you want to

check to make sure it's the window. If it is, you call *Update_IV_Window()* to draw to the window. Here's *Handle_Update():*

```
void  Handle_Update( void )
{
   WindowPtr  the_window;

   the_window = ( WindowPtr )The_Event.message;

   if ( the_window == IV_Window_Ptr )
      Update_IV_Window();
}
```

Update_IV_Window() examines the *Display_Hardware_Flag* to determine whether hardware or software information is to be displayed in the window. You must have the correct pair of routines to get the information and then display it.

```
void  Update_IV_Window( void )
{
   GrafPtr    old_port;
   WindowPtr  the_window;

   GetPort( &old_port );
   SetPort( IV_Window_Ptr );

   TextFont(systemFont);
   TextSize(12);

   BeginUpdate( IV_Window_Ptr );

      Draw_Mac_Picture();
      Draw_Owner_Information();
      Draw_System_Info_Headings();

      if ( Display_Hardware_Flag == TRUE )
      {
         Get_Hardware_Information();
         Display_Hardware_Information();
      }
      else
      {
         Get_Software_Information();
         Display_Software_Information();
```

```
        }

    EndUpdate( IV_Window_Ptr );

    SetPort( old_port );
}
```

A better approach might be to call each of the two routines that get the information a single time, perhaps in an initialization routine in Segment 3. Then they needn't be called each time an update was performed; the display routines could just redraw the information each time. Good thinking!

However... Apple's big on *Gestalt()*. It's been developed in such a way that it can, and will be, continually expanded to allow access to more features of your Macintosh hardware and system software. What if one of those features is a check of available free RAM, which is constantly changing? Or some other parameter that is dynamic? If this program were to display more machine features (and if it were a real application, it would), you might display some of these features that change "on-the-fly." Now you're ready for that!

Getting machine information, and error handling

Get_Hardware_Information() uses *Gestalt()* to get machine information. Last chapter's InnerView program gave a good example of the powerful *Gestalt()* function. This chapter's new *InnerViewII* uses it in a similar fashion. Here you use the error-handling routine to verify that the call worked.

You start out the routine with the initialization of the local *Boolean* variable *failed* to false. After each call to *Gestalt()* you check the result err for an error. If the call failed, *err* will have a value other than *noErr*. If that happens you set *failed* to true. Here's how one of the three calls to *Gestalt()* is handled:

```
void  Get_Hardware_Information( void )
{
   OSErr    err;
   long     response;
   Boolean  failed = FALSE;

   err = Gestalt( gestaltProcessorType, &response );
   if ( err == noErr )
      Mac_CPU = response;
   else
   {
      Mac_CPU = GESTALT_ERR_TYPE;
      failed = TRUE;
   }
```

When the routine is done, you check to see if failed was set to true. If one or more of the calls didn't work, failed will equal true. Now it's time to post an error message:

```
if ( failed == TRUE )
   Post_Error_Message(ERR_CALL_TO_GESTALT_FAIL, DONT_TERMINATE_ERROR);
```

Why wait till the end of the routine to post the message? Why not check *failed* after each call? In the unlikely case that all three calls didn't work, the alert (with the same "gestalt failed" message) would appear three times. This doesn't give the user much extra information and might lead him to believe there's some other problem, such as the program hanging in a loop.

The routine that gets software information, *Get_Software_Information()* works in the same way as this routine. How that routine handles the system version requires some explanation. I do that next.

Bits, masks, and Str255

One of the *Gestalt()* calls made in *Get_Software_Information()* returns the system version in hexadecimal format. That's 0x0, followed by the version. For example, System 7.1.0 would be returned as 0x0710. *InnerViewII* needs to display the hex number as a string. You can't just call *NumToString()*, because that routine first converts the hexadecimal value to decimal, and that's not what you want.

You need to extract each of the last three hex digits from the number, make each an individual character, then piece them together into a string to display. Here's the code that does that:

```
for ( i=1; i <= 3; i++ )
{
   temp = response;
   if ( i == 1 )
      digit = ( temp &= 0x0F00 ) / 0x0100;
   else if ( i == 2 )
      digit = ( temp &= 0x00F0 ) / 0x0010;
   else
      digit = ( temp &= 0x000F ) / 0x0001;
   digit += ASCII_ZERO;
   Mac_Sys_Str[i] = digit;
}
Mac_Sys_Str[0] = 3;
```

The first step is to extract the last three digits. For that you use the bit-wise *&* operator, which operates on a value bit-by-bit. If you use it in conjunction with a mask, you can pull out a single digit from the hex number. The first pass through the above *for* loop looks like this:

```
temp = response;
if ( i == 1 )
   digit = ( temp &= 0x0F00 ) / 0x0100;
```

Here the mask is 0x0F00. When "ANDED" with another hex number, the only bits that pass through into the result are those that have a 1 in the bit positions corresponding to the 1's in the mask. Figure 9–52 shows what takes place in *temp &= 0x0F00*. Remember, that's the same as *temp = temp & 0x0F00*.

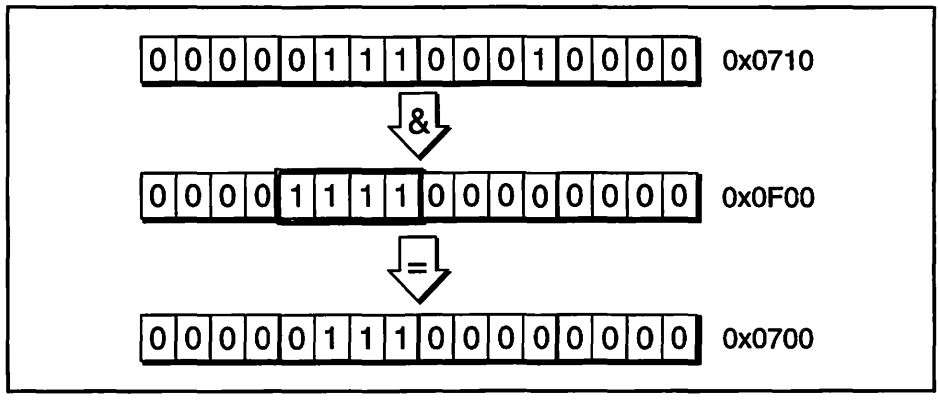


Figure 9–52. *Using a mask and the & operator*

Figure 9–52 shows that the result of the *&* operation is a hex number that has masked out every digit except the 7. The next step is to divide by 0x0100 to extract the 7. However, you don't want the number 7; you want the ASCII character for it. This is the step that does that:

```
digit += ASCII_ZERO;
```

ASCII_ZERO is defined to be 48. Adding 48 to any digit gives its ASCII equivalent, as the abbreviated ASCII table in Figure 9–53 shows.

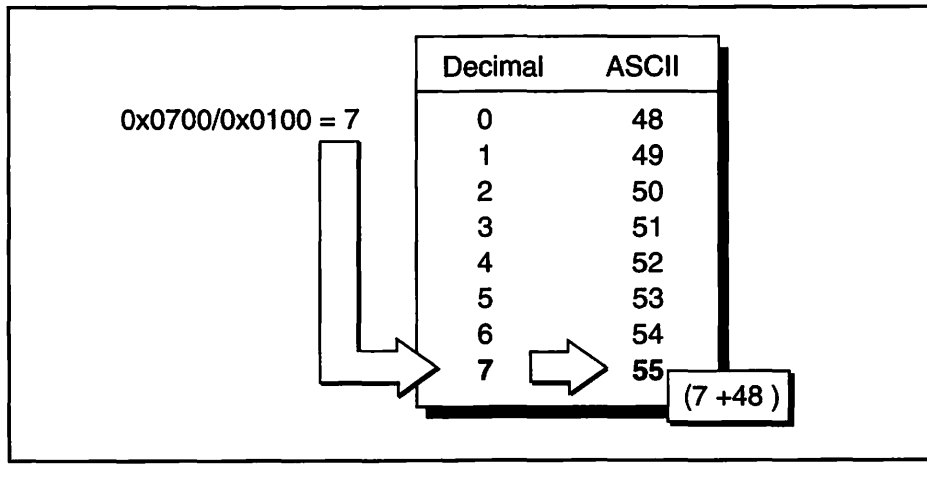


Figure 9–53. *Decimal to ASCII conversion*

The final step is to insert the digit into a string. You use the loop counter as the placeholder:

```
Mac_Sys_Str[i] = digit;
```

The *Str255* type, which *Mac_Sys_Str* is declared to be, has the format shown in Figure 9–54. The first array element is the number of characters in the string. The remaining elements are the characters themselves.

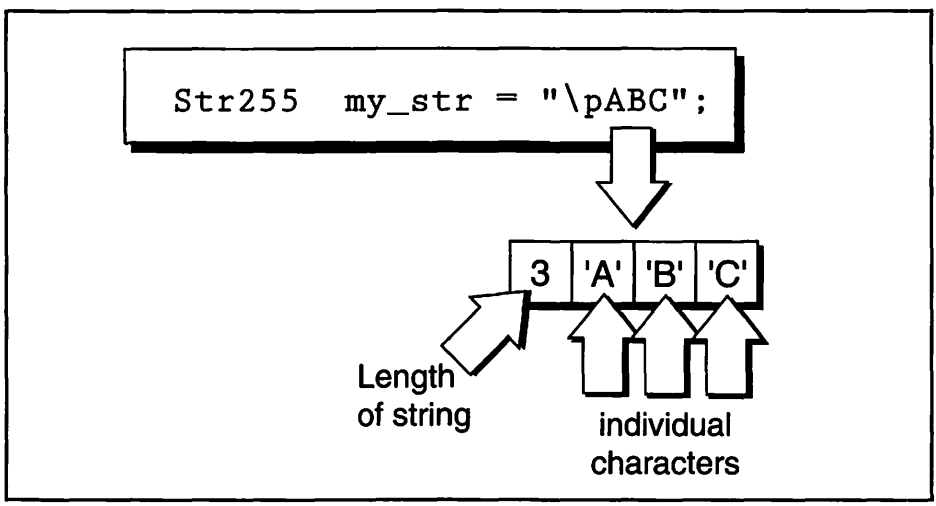


Figure 9–54. *Format of a variable of Str255 type*

You know that this system version string will always be three characters in length. You set the string length with the following line:

```
Mac_Sys_Str[0] = 3;
```

Displaying machine information

After each call to *Gestalt()* in *Get_Hardware_Information()*, a global variable was set to keep track of the result of the call. Now, in *Display_Hardware_Information()*, it's time to use the values of those flags. Here's the technique:

```
void  Display_Hardware_Information( void )
{
   Str255   the_str;
```

```
short    the_list;
short    the_index;

switch ( Mac_CPU )
{
   case gestalt68000:
      the_index = 1;
      break;
   case gestalt68010:
      the_index = 2;
      break;
   case gestalt68020:
      the_index = 3;
      break;
   case gestalt68030:
      the_index = 4;
      break;
   case gestalt68040:
      the_index = 5;
      break;
   default:
      the_index = 6;
      break;
}
the_list = CPU_TYPE_STR_LIST;
GetIndString( the_str, the_list, the_index );
MoveTo( RESULT_X, HEADING_Y + LINE_HEIGHT );
DrawString( the_str );
```

Here you use a *switch* statement that compares the global variable *Mac_CPU* with the various constants that describe the feature you're looking at. These constants come from the header file *GestaltEqu.h*. (Many of them are listed in Appendix C.) Within the switch you set an index that will be the index into the 'STR#' list holding the text to be displayed in the window. Figure 9–55 shows what happens to the above call to *GetIndString()* when *InnerViewII* runs on a Macintosh with a 68000 CPU.

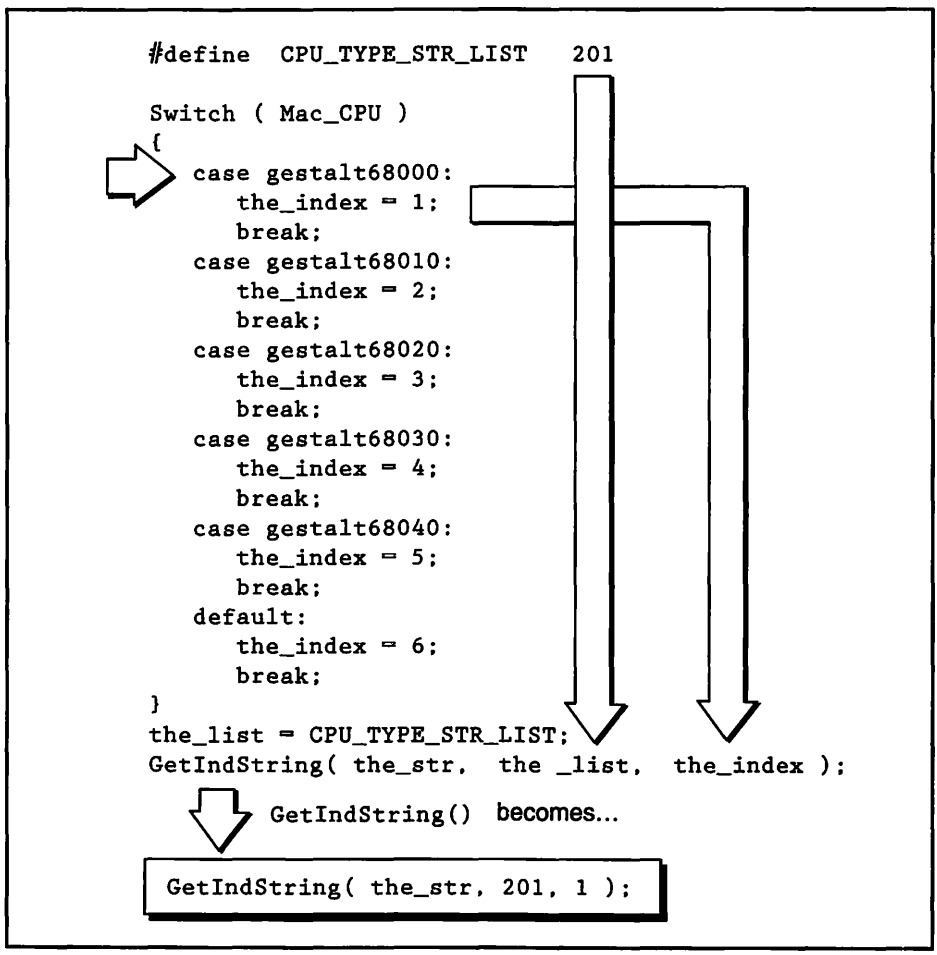


Figure 9–55. *The parameters to GetIndString()*

What is the string returned to the program by a call to *GetIndString(the_str, 201, 1)*? The string "68000". That's the first string (index = 1) in 'STR#' 201, as shown in Figure 9–56.

InnerViewII also relies on 'STR#' strings to write the heading for the window in *Draw_System_Info_Headings()* and to draw the name of the owner in *Draw_Owner_Information()*.

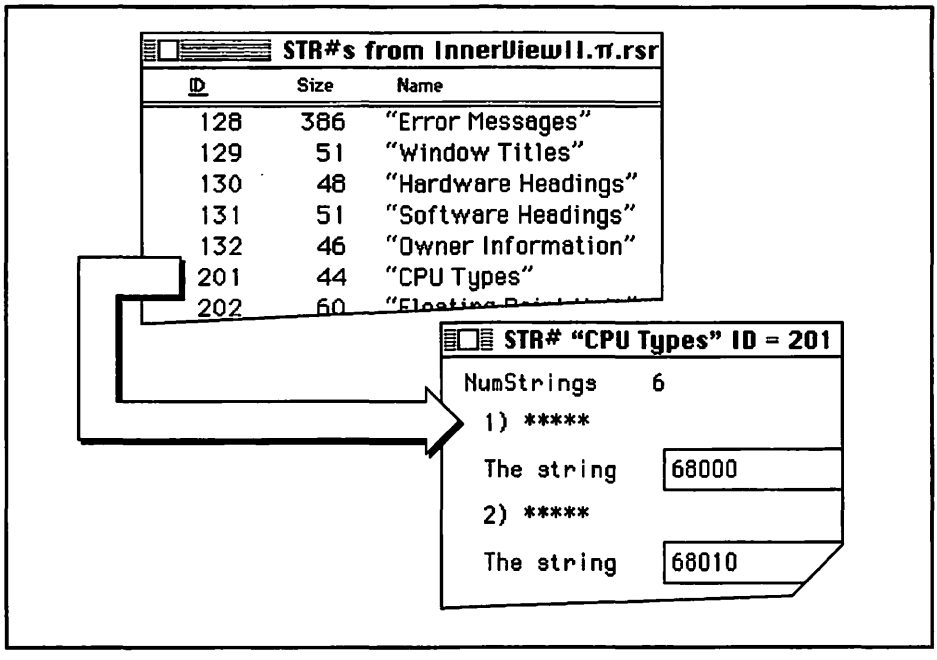


Figure 9-56. *Where the string comes from in a call to* GetIndString()

Displaying a picture

You've seen how to display a picture several times in this book. Here, the difference from previous efforts is that there is no assumption that the drawing succeeded. If memory is scarce, and the 'PICT' resource holding the picture is large, a call to *DrawPicture()* might fail. You can make a check for that by examining the *PicHandle* returned by the call. If the handle is nil, you call the error-handling routine. This test may be overkill here, because the 'PICT' is only about 5 K in size. But for large, color pictures, it is realistic.

```
the_pict = GetPicture( pict_id );
if ( the_pict == NIL )
   Post_Error_Message(ERR_PICT_MEM_ALLOCATE_FAIL, DONT_TERMINATE_ERROR);
```

Sure, I'm proud of the picture of the Mac and the magnifying glass that I drew up in a paint program. Heck, I've used it in enough examples! But I'm not so vain that I think the program should cease if my masterpiece isn't displayed! If a call to *DrawPicture()* fails, the picture won't

be displayed. But the program will go on. That's why I pass *Post_Error_Message()* a value of *DONT_TERMINATE_ERROR*. You can be liberal with the display of error messages. You want to prevent a frozen Mac, and you want to let the user know he's missing out on something such as a picture not being displayed. But use your discretion when deciding if an error is serious enough to warrant a call to *ExitToShell()*.

Chapter Summary

Memory management—the allocation, movement, tracking, and removing of objects in memory—requires some planning on your part. Familiarity with how the Memory Manager works will aid you in writing programs free of memory-related bugs.

Nonrelocatable blocks, which cannot be moved during compaction, are created through calls to *NewPtr()*. By reserving a store of memory early in your program, you can set aside a block low in memory for nonrelocatable blocks. A nonrelocatable block that is at the bottom of memory keeps it from fragmenting your program's application heap.

A master pointer is used every time a relocatable block is created with a call to *NewHandle()*. While the block used by the call to *NewHandle()* is relocatable, the memory occupied by the master pointer isn't. By calling the Toolbox routine *MoreMasters()* early in your program you can reserve a block of master pointers low in memory.

When your program starts up, its heap is set to a small size. The heap will expand as the program executes. A more efficient use of memory involves enlarging the heap to its maximum size as your program starts. A call to the Toolbox routine *MaxApplZone()* does just that.

The code for a Macintosh program is contained in segments, or parcels, no larger than 32 K each. By spending time determining a strategy for segmenting your code, you can make your program's memory usage more efficient. The Toolbox routine *UnloadSeg()* marks a segment purgeable. The routine allows the Memory Manager to remove the segment from memory if memory becomes tight and the segment is not in use.

Each program has its own memory partition. You can set that partition size using a 'SIZE' resource. By not setting an overly-large partition size, you give your programs users access to more of their computer's RAM. Computer memory is an abstract concept; it is difficult to determine just how much memory a program requires. Using a utility like *Swatch* assists you in understanding how your program uses its allotted partition.

If a call to a Toolbox routine fails, an error occurs. For example, a call to *GetPicture()* may attempt to load a 'PICT' resource that is too large to fit in available memory. You should always examine your code for potential error conditions and call an error-handling routine in the event of a failure.

An error-handling routine should display a descriptive message that will help the user remedy the situation. Additionally, if the error is severe the program should exit to the Finder rather than risk a crash that will freeze the user's system.

A Macintosh C Data Types

ANSI C data types, such as *int*, *float*, and *char*, all exist in the Macintosh programming world. But to meet the special GUI needs of the Macintosh, Apple has created several new data types. These types allow access to the Toolbox, provide you with a means to create and work with the graphical user interface, and give you the resource to work within Apple's memory addressing scheme. Many of these data types are defined alphabetically in this appendix.

cGrafPort

The color version of a *GrafPort*. See GrafPort.

ControlHandle

The push buttons, radio buttons, and check boxes found in dialog boxes are controls. To work with them, Toolbox routines use handles to them called *ControlHandles*.

Cursor

The data type that represents a 16-by-16 bit image that defines a cursor. The on-screen cursor is set to the arrow cursor by a call to *InitCursor()*. To access the other four system-defined cursors, use the constants *iBeamCursor, crossCursor, plusCursor, and watchCursor.*

CursHandle

Cursors are stored as 'CURS' resources and are accessed by Toolbox routines that return, and expect as a parameter, the cursor handle *CursHandle.*

DialogPtr

A pointer to another Macintosh data type, the *DialogRecord*. The *DialogRecord* holds information about a dialog box. You access this information via Toolbox calls that require a *DialogPtr* rather than the *DialogRecord* itself. Many Toolbox routines that work with *WindowPtrs* also work with a *DialogPtr* as the parameter.

DialogRecord

A structure that holds information about a single dialog box—descriptive information needed by the Dialog Manager. You seldom need to work directly with a *DialogRecord*. Instead, you access information indirectly through Toolbox routines that use a *DialogPtr*, a pointer to a *DialogRecord.*

EventRecord

EventRecord holds information about a single event. An *EventRecord* is created for every event that occurs. These *EventRecords* are held in an event queue. Unlike some Macintosh data types, that you deal with through the use of pointers or handles, you work with events directly through the record itself.

grafPort

A graphics port is a drawing environment that defines how text and shapes will be drawn. So that it can display unique text or shape styles, each window has its own graphics port. A *grafPort* is the Macintosh data type that

holds this information about a graphics port. To access information within a graphics port you use a pointer to a *grafPort*, rather than the *grafPort* itself.

grafPtr

A pointer to a *grafPort*. A *grafPort* is the data structure that holds information about a graphics port. See *grafPort*.

Handle

A pointer to a master pointer. A master pointer keeps track of the location of a relocatable block in the application's heap. Some Toolbox functions return a *Handle* to your program. To make use of this generic *Handle* in future Toolbox calls, you may have to typecast it to a specific type of handle, such as a *ControlHandle*.

MenuHandle

A handle to a menu record. A menu record holds information about a single menu—descriptive information needed by the Menu Manager. Toolbox routines that work with menus use *MenuHandles* rather than the menu record itself.

PatHandle

A data type. Patterns can be created and stored in 'PAT ' resources. Toolbox routines that work with *Patterns* obtained from a resource file use the *PatHandle* data type.

Pattern

An 8-by-8 bit image that defines a design that can be repeated to fill an area of any given size. There are five system Patterns defined by the constants *white, ltGray, gray, dkGray,* and *black*.

PicHandle

A handle type. Pictures, or 'PICT' resource types, are accessed through a handle of type *PicHandle*. Toolbox routines that work with pictures will expect a *PicHandle* as a parameter.

Point

Any pixel on the Macintosh screen can be referred to by a pair of coordinates. That data type *Point* holds one such pair.

Rect

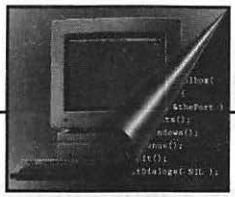
A rectangle. It is the Macintosh data type that is used as a basis for drawing rectangles, ovals, and round rectangles. The coordinates that make up a rectangle's upper-left corner and the two that make up its lower-right corner define a rectangle. The Macintosh data type that holds this information is the *Rect*.

WindowPtr

A pointer to another Macintosh data type, the *WindowRecord*. The *WindowRecord* holds information about a window. You'll access this information via Toolbox calls that require a *WindowPtr* rather than the *WindowRecord* itself.

WindowRecord

A structure that holds information about a single window—descriptive information needed by the Window Manager. You seldom need to work directly with a *WindowRecord*. Instead, you access information indirectly through Toolbox routines that use a *WindowPtr*, a pointer to a *WindowRecord*.



B Determining a Trap's Type

Chapter 8, The Varying Mac, thoroughly covered the concept of traps. For you trap fanatics, here's a little more.

If your program is running on a computer that has System 6.0.4 or later, you can use the *Gestalt()* function to determine the availability of a trap quickly and easily. The *Gestalt()* routine is discussed in Chapter 8. If you're on a machine that is pre-1989, you have to use the *NGetTrapAddress()* function in place of Gestalt().

If you're using *NGetTrapAddress()*, and you know the trap you are looking for is a Toolbox trap (as opposed to an Operating System trap,) you can simply make the comparison to the Unimplemented trap. Here's the example used in Chapter 8:

```
Boolean  Color_Wind_Available;

Color_Wind_Available = (NGetTrapAddress(_Unimplemented, ToolTrap)
                    !=  NGetTrapAddress(_GetNewCWindow, ToolTrap));
```

If, however, you're writing a program that is to run on a pre-1989 Macintosh, and you don't know the type of the trap, *ToolTrap* or *OSTrap*, you'll need to include extra code in your program. Below is the necessary code. The routine *Trap_Is_Present()* and the two routines that it calls are summarized here.

Get_Trap_Type()

The setting of one bit of a trap—bit 11—determines whether the trap is a Toolbox trap (*ToolTrap*) or an Operating System trap (*OSTrap*). This routine performs an & operation on this one bit to determine if it is set or not.

Num_Tool_Traps()

Macintosh models may have one of two different sized trap tables. This routine uses the trap for the *InitGraf()* routine, present on all Macs, to determine if the Toolbox has 512 (0x200) Toolbox traps or 1024 (0x400) Toolbox traps.

Trap_Is_Present()

This routine makes use of the *Get_Trap_Type()* and *Num_Tool_Traps()* routines and the Toolbox routine *NGetTrapAddress()* to determine if the trap you've passed in is present.

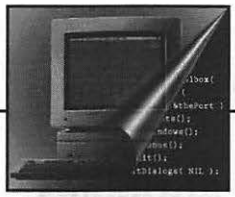
Now, the code. For in-depth trap-checking, copy the following three routines to your source code—even if you don't fully understand them! Then, to check for the availability of a trap, simply call *Trap_Is_Present()*, passing in the trap to check for. Following the routines is an example of a call to *Trap_Is_Present()*.

```
Boolean   Trap_Is_Present( short );
TrapType  Get_Trap_Type( short );
short     Num_Tool_Traps( void );

Boolean  Trap_Is_Present( short the_trap )
{
   TrapType  the_type;
   Boolean   present;
```

```
    the_type = Get_Trap_Type( the_trap );

    if ( ( the_type == ToolType ) &&
       ( ( the_trap &= 0x07FF ) >= Num_Tool_Traps() ) )
       present = FALSE;
    else
       present = ( NGetTrapAddress( _Unimplemented, ToolTrap ) !=
                   NGetTrapAddress( the_trap, the_type ) );

    return present;
}


TrapType  Get_Trap_Type( short the_trap )
{
    if ( ( the_trap & 0x0800 ) == 0 )
       return ( OSTrap );
    else
       return ( ToolTrap );
}


short  Num_Tool_Traps( void )
{
    if ( NGetTrapAddress( 0xA86E, ToolTrap ) ==
         NGetTrapAddress( 0xAA6E, ToolTrap ) )
       return ( 0x200 );
    else
       return ( 0x400 );
}
```

Here's an example that checks for the presence of *WaitNextEvent()*. If its not available, an alert will be posted.

```
#include <Traps.h>

#define    WNE_ERR_ALRT    128
#define    NIL_PTR         0L
...
...
if (Trap_Is_Present(_WaitNextEvent) == FALSE)
   Alert(WNE_ERR_ALRT, NIL_PTR);
```



C Gestalt Definitions

Chapter 8 gave several examples for obtaining information about a Macintosh using the *Gestalt()* function. Here are several more selector codes that yield system software and hardware information. If you plan on expanding Chapter 9's *InnerViewII* example program into a more useful utility, you'll want to add *Gestalt()* calls that include many of these selector codes.

Use any of the selector codes as shown in Chapter 8. Here's an example:

```
#include <GestaltEqu.h>

OSErr  err;
long   response;

err = Gestalt(gestaltQuickdrawVersion, &response);

if ((err == noErr) && (response == gestaltOriginalQD))
   DrawString("\pYou have the original version of QuickDraw.");
```

Addressing Mode Attributes

Selector code

```
gestaltAddressingModeAttr
```

Response parameter

```
gestalt32BitAddressing = 0   /* using 32-bit addressing mode  */
gestalt32BitSysZone    = 1   /* 32-bit compatible system zone */
gestalt32BitCapable    = 2   /* 32-bit capable machine        */
```

Apple Events Attributes

Selector code

```
gestaltAppleEventsAttr
```

Response parameter

```
gestaltAppleEventsPresent = 0 /* true if Apple Events present */
```

AppleTalk Version

Selector code

```
gestaltAppleTalkVersion
```

Response parameter

Returns version number of installed AppleTalk driver.

A/UX Version

Selector code

```
gestaltAUXVersion
```

Response parameter

Returns version number of A/UX if it is currently executing.

Easy Access Attributes

Selector code

```
gestaltEasyAccessAttr
```

Response parameter

```
gestaltEasyAccessOff    = 0   /* Easy Access present, but off */
gestaltEasyAccessOn     = 1   /* Easy Access On               */
gestaltEasyAccessSticky = 2   /* Easy Access Sticky           */
gestaltEasyAccessLocked = 3   /* Easy Access Locked           */
```

Font Manager Attributes

Selector code

```
gestaltFontMgrAttr
```

Response parameter

```
gestaltOutlineFonts = 0    /* true if Outline Fonts supported */
```

Floating-Point Unit Type

Selector code

```
gestaltFPUType
```

Response parameter

```
gestaltNoFPU    = 0        /* no FPU             */
gestalt68881    = 1        /* 68881 FPU          */
gestalt68882    = 2        /* 68882 FPU          */
gestalt68040FPU = 3        /* 68040 built-in FPU */
```

Gestalt Version

Selector code

```
gestaltVersion
```

Response parameter

Returns the current version. As of this writing the current version is 1, returned as $0001.

Hardware Attributes

Selector code

```
gestaltHardwareAttr
```

Response parameter

```
gestaltHasVIA1          = 0     /* VIA1 exists              */
gestaltHasVIA2          = 1     /* VIA2 exists              */
gestaltHasASC           = 3     /* Apple Sound Chip exists */
gestaltHasSCC           = 4     /* SCC exists               */
gestaltHasSCSI          = 7     /* SCSI exists              */
gestaltHasSoftPowerOff  = 19    /* has software power off   */
gestaltHasSCSI961       = 21    /* 53C96 SCSI controller    */
gestaltHasSCSI962       = 22    /* 53C96 SCSI controller    */
gestaltHasUniversalROM  = 24    /* has a Universal ROM      */
```

Help Manager Attributes

Selector code

```
gestaltHelpMgrAttr
```

Response parameter

```
gestaltHelpMgrPresent = 0  /* true if help mgr is present */
```

Keyboard Type

Selector code

```
gestaltKeyboardType
```

Response parameter

```
gestaltMacKbd           = 1    /* Mac keyboard              */
gestaltMacAndPad        = 2    /* Mac keyboard w/pad        */
gestaltMacPlusKbd       = 3    /* MacPlus keyboard          */
gestaltExtADBKbd        = 4    /* extended ADB keyboard     */
gestaltStdADBKbd        = 5    /* standard ADB keyboard     */
gestaltPrtblADBKbd      = 6    /* portable ADB keyboard     */
gestaltPrtblISOKbd      = 7    /* portable ISO keyboard     */
gestaltStdISOADBKbd     = 8    /* standard ISO ADB keyboard */
gestaltExtISOADBKbd     = 9    /* extended ISO ADB keyboard */
gestaltADBKbdII         = 10   /* ADB keyboard II           */
gestaltADBISOKbdII      = 11   /* ADB ISO keyboard II       */
gestaltPwrBookADBKbd    = 12   /* Powerbook ADB keyboard    */
gestaltPwrBookISOADBKbd = 13   /* Powerbook ISO ADB keyboard */
Logical RAM Size
```

Selector code

```
gestaltLogicalRAMSize       /* logical ram size */
```

Response parameter

Returns the amount of logical memory available.

Low Memory Area

Selector code

```
gestaltLowMemorySize        /* size of low memory area */
```

Response parameter

Returns the size, in bytes, of the low-memory area. This area is used for vectors, global variables, and dispatch tables.

Memory Management Unit Type

Selector code

```
gestaltMMUType
```

Response parameter

```
gestaltNoMMU     = 0        /* no MMU                    */
gestaltAMU       = 1        /* address management unit */
gestalt68851     = 2        /* 68851 PMMU                */
gestalt68030MMU = 3        /* 68030 built-in MMU        */
gestalt68040MMU = 4        /* 68040 built-in MMU        */
```

Processor Type

Selector code

```
gestaltProcessorType
```

Response parameter

```
gestalt68000 = 1           /* Motorola 68000 CPU */
gestalt68010 = 2           /*          68010 CPU */
gestalt68020 = 3           /*          68020 CPU */
gestalt68030 = 4           /*          68030 CPU */
gestalt68040 = 5           /*          68040 CPU */
```

QuickDraw Version

Selector code

```
gestaltQuickdrawVersion
```

Response parameter

```
gestaltOriginalQD = 0x000     /* original 1-bit QuickDraw     */
gestalt8BitQD     = 0x100     /* 8-bit color QuickDraw        */
gestalt32BitQD    = 0x200     /* 32-bit color QuickDraw       */
gestalt32BitQD11  = 0x210     /* 32-bit color QuickDraw v1.1  */
gestalt32BitQD12  = 0x220     /* 32-bit color QuickDraw v1.2  */
gestalt32BitQD13  = 0x230     /* 32-bit color QuickDraw v1.3  */
```

Physical RAM Size

Selector code

```
gestaltPhysicalRAMSize
```

Response parameter

Returns the number of bytes of phgysical RAM currently installed.

Sound Attributes

Selector code

```
gestaltSoundAttr
```

Response parameter

```
gestaltStereoCapability    = 0  /* stereo compatabile hardware    */
gestaltStereoMixing        = 1  /* external speaker stereo mixing */
gestaltSoundIOMgrPresent   = 3  /* Sound I/O Manager is present   */
gestaltBuiltInSoundInput   = 4  /* built-in Sound Input hardware  */
gestaltHasSoundInputDevice = 5  /* Sound Input device available   */
```

Virtual Memory Attributes

Selector code

```
gestaltVMAttr
```

Response parameter

```
gestaltVMPresent = 0    /* true if virtual memory is present */
```

The remaining selector codes are for informational use only. Don't base programming decisions on the returned response. Chapter 8 gives more information on this.

Machine Type

Selector code

```
gestaltMachineType
```

Response parameter

```
kMachineNameStrID   = -16395
gestaltClassic      = 1
gestaltMacXL        = 2
gestaltMac512KE     = 3
gestaltMacPlus      = 4
gestaltMacSE        = 5
gestaltMacII        = 6
gestaltMacIIx       = 7
gestaltMacIIcx      = 8
gestaltMacSE030     = 9
gestaltPortable     = 10
gestaltMacIIci      = 11
gestaltMacIIfx      = 13
gestaltMacClassic   = 17
gestaltMacIIsi      = 18
gestaltMacLC        = 19
gestaltQuadra900    = 20
gestaltPowerBook170 = 21
gestaltQuadra700    = 22
gestaltClassicII    = 23
gestaltPowerBook100 = 24
gestaltPowerBook140 = 25
```

Machine Icon

Selector code

```
gestaltMachineIcon
```

Response parameter

Returns an icon family resource ID for the type of Macintosh.

ROM Size

Selector code

```
gestaltROMSize
```

Response parameter
Returns the size of the installed ROM.

ROM Version

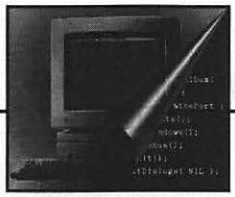
Selector code
```
gestaltROMVersion
```

Response parameter
Returns the version number of the installed ROM.

System Version

Selector code
```
gestaltSystemVersion
```

Response parameter
Returns the version number of the active System file.



D Toolbox Routine Summary

This appendix summarizes the Toolbox calls used throughout this book. The calls are divided into the following eight sections:

- QuickDraw
- Events
- Windows
- Dialogs
- Menus
- Memory
- Utilities
- Sound

QuickDraw

This section describes many of the important QuickDraw routines found in the Toolbox.

There are additional constants and data structures listed in Apple's *QuickDraw.h* header file. You *do not* have to *#include* this file in your projects. Both the THINK C *MacHeaders* and the Symantec C++ *MacHeaders++* files include this header, and many others. By default, your THINK C and Symantec C++ projects contain *MacHeaders* or *MacHeaders++*.

Constants

```
#define    systemFont      0
#define    applFont        1
#define    newYork         2
#define    geneva          3
#define    monaco          4
#define    venice          5
#define    london          6
#define    athens          7
#define    sanFran         8
#define    toronto         9
#define    cairo          11
#define    losAngeles     12
#define    times          20
#define    helvetica      21
#define    courier        22
#define    symbol         23
#define    mobile         24
```

TextFont() changes the font that subsequent calls to *DrawString()* uses. Pass *TextFont()* any of the above constants to set drawing to that font.

```
#define    normal          0
#define    bold            1
#define    italic          2
#define    underline       4
#define    outline         8
#define    shadow       0x10
```

```
#define    condense    0x20
#define    extend      0x40
```

TextStyle() sets the style of text drawn by *DrawString()*. Pass *TextStyle()* any of the above values or add any number of *Styles* together for a combined effect.

Global Variables

```
Pattern  dkGray;

Pattern  ltGray;

Pattern  gray;

Pattern  black;

Pattern  white;
```

The system defines five *Patterns* for your use in calls such as *PenPat()* or *FillRect()*. You can use any of the five variables without declaring them in your program.

Data Structures

```
struct GrafPort
{
   short      device;
   BitMap     portBits;
   Rect       portRect;
   RgnHandle  visRgn;
   RgnHandle  clipRgn;
   Pattern    bkPat;
   Pattern    fillPat;
   Point      pnLoc;
   Point      pnSize;
   short      pnMode;
   Pattern    pnPat;
   short      pnVis;
   short      txFont;
   Style      txFace;
   char       filler;
   short      txMode;
```

```
    short       txSize;
    Fixed       spExtra;
    long        fgColor;
    long        bkColor;
    short       colrBit;
    short       patStretch;
    Handle      picSave;
    Handle      rgnSave;
    Handle      polySave;
    QDProcsPtr  grafProcs;
};

typedef  struct  GrafPort  GrafPort;
typedef  GrafPort  *GrafPtr;
```

A *GrafPort* is the drawing environment of a window. Each window has its own *GrafPort*. The fields within a *GrafPort* are changed through the use of Toolbox calls rather than direct manipulation.

Graphic Ports

```
void  SetPort( GrafPtr  the_port );
```

SetPort() makes the graphics port pointed to by *the_port* the current port. Subsequent drawing operations will be performed in this port. Call *SetPort()* before first drawing to a port to ensure that graphics operations are drawn to the proper window. Before calling *SetPort()*, call *GetPort()* to save the current port so that it can be restored later.

```
void  GetPort( GrafPtr  *the_port );
```

GetPort() gets the current port and saves a pointer to it in *the_port*. Before drawing to a port, call *GetPort()* to save the current port, then call *SetPort()* to set the port to the new port.

Graphics Pen

```
void  GetPenState( PenState  *pen_state );
```

GetPenState() gets the current state of the pen and stores it in *pen_state*. After making location, size, or pattern changes to the pen, you can restore the previous pen state with a call to *SetPenState()*.

```
void  SetPenState( PenState  *pen_state );
```

Before making changes to the state of the graphics pen you can call *GetPenState()* to save the current state in *pen_state*. Then, after drawing is complete, call *SetPenState()*. Pass the same *pen_state* to restore the pen to its previous condition.

```
void  PenPat( Pattern the_pattern );
```

PenPat() sets the pattern used by the graphics pen to *the_pattern*. All subsequent drawing operations performed in the current graphics port will use this pattern until *PenPat()* is again called. Use *GetPattern()* to load a 'PAT ' resource for use by *PenPat()*, or use one of the standard *Patterns* defined as global variables and listed under the Global Variables heading of this section.

```
void  PenPixPat( PixPatHandle pat_handle );
```

PenPixPat() sets the pattern used by the graphics pen to the pattern accessed through *pat_handle*. All subsequent drawing operations performed in the current graphics port will use this pattern until *PenPixPat()* is again called. Call *GetPixPat()* to load a color 'ppat' resource for use by *PenPixPat()*.

```
void  PenSize( short  width,
               short  height );
```

Set the width and height of the graphics pen with a call to *PenSize()*. The *width* and *height* are the pixel dimensions the pen will acquire. All subsequent lines drawn with the pen will be drawn in this size.

```
void  PenNormal( void );
```

To restore the pen to its default settings, call *PenNormal()*. *PenNormal()* sets the pen's size to (1,1) and its pattern to *black*.

```
void  MoveTo( short  horiz,
              short  vert );
```

MoveTo() moves the pen to the horizontal pixel coordinate *horiz* and the vertical pixel coordinate *vert*. The origin is the left top corner of the current port. No drawing is performed.

```
void  Move( short  horiz,
            short  vert );
```

Move() moves the pen *horiz* pixels in the horizontal direction and *vert* pixels in the vertical position from the pen's current position. A negative *horiz* value moves the pen to the left. A negative *vert* value moves the pen up. No drawing is performed.

```
void  LineTo( short  horiz,
              short  vert );
```

LineTo() draws a line to the horizontal pixel coordinate *horiz* and the vertical pixel coordinate *vert*. The origin is the left top corner of the current port.

```
void  Line( short  horiz,
            short  vert );
```

Line() draws a line *horiz* pixels in the horizontal direction and *vert* pixels in the vertical position from the pen's current position. A negative *horiz* value draws a line to the left. A negative *vert* value draws a line up.

Drawing Text

```
void  GetIndString( Str255  the_str,
                    short   str_list_ID,
                    short   index );
```

GetIndString() loads a string into *the_str* from the 'STR#' list with an ID of *str_list_ID*. From this list *GetIndString()* selects the *index* string in the list; e.g. if *index = 2*, the second string in the list will be loaded. Once loaded, *the_str* can be used as any other *Str255* variable.

```
void  TextFont( short  font_num );
```

TextFont() sets the font to the font number *font_num*. All subsequent text will be drawn in this font. Many fonts are defined by constants given under the Constants heading of this section.

```
void  TextFace( Style  face );
```

TextFace() sets the style of text to *face*. The style can be one *Style* or a combination of *Styles*. All subsequent text will be drawn in this style. See the Constants heading in this section for a listing of the available *Styles*.

```
void  TextSize( short  size );
```

Set the size of text with a call to *TextSize()*. The *size* is given in points, where approximately 72 points equals one inch. All subsequent text will be drawn in this size.

```
void  DrawChar( short  ch );
```

DrawChar() draws a single character *ch* to the current port. The current font, style, and size are used. The starting location of the character is the current position of the graphics pen.

```
void  DrawString( Str255  the_str );
```

DrawString() draws string *the_str* to the current port. The current font, style, and size are used. The starting location of the character is the current position of the graphics pen.

Patterns

```
PatHandle  GetPattern( short  pattern_ID );
```

GetPattern() returns a *PatHandle* to the 'PAT ' resource with the ID of *pattern_ID*. Once you've obtained a *PatHandle*, dereference it twice and then use it as a *Pattern* type in QuickDraw calls, such as *PenPat()*.

```
PixPatHandle  GetPixPat( short  ppat_ID );
```

GetPixPat() works like *GetPattern()*. *GetPixPat()* returns a *PixPatHandle* to the 'ppat' resource with the ID of *ppat_ID*. Once you've obtained a *PixPatHandle*, use it in QuickDraw calls, such as *PenPixPat()*. Color QuickDraw routines that work with color patterns accept handles to them—you do not have to dereference it.

Drawing Shapes

```
void  SetRect( Rect  *the_rect,
               short  left,
               short  top,
               short  right,
               short  bottom );
```

SetRect() sets the boundaries of rectangle *the_rect*. The coordinates of the rectangle use the current graphics port's left top corner as the origin.

Always use *SetRect()* to establish a rectangle before performing shape-drawing operations involving a rectangle, oval, or round rectangle. *SetRect()* does not display a rectangle.

```
void  FrameRect( Rect  *the_rect );
```

FrameRect() frames rectangle *the_rect*. Before framing, establish the boundaries of *the_rect* with a call to *SetRect()*. *FrameRect()* does not fill in the rectangle, it merely outlines it with a frame.

```
void  PaintRect( Rect  *the_rect );
```

PaintRect() fills the rectangle *the_rect* with the current pen pattern. Call *SetRect()* to establish the boundaries of *the_rect*.

```
void  FillRect( Rect    *the_rect,
                Pattern  the_pat );
```

FillRect() fills the rectangle *the_rect* with the pattern *the_pat*. The current pen pattern is unaffected by the call to *FillRect()*. Call *SetRect()* to establish the boundaries of *the_rect*.

```
void  EraseRect( Rect  *the_rect );
```

EraseRect() fills rectangle *the_rect* with the background pattern, which is usually *white*. Call *SetRect()* to establish the boundaries of *the_rect*.

```
void  InvertRect( Rect  *the_rect );
```

InvertRect() changes the state of each pixel in rectangle *the_rect*. All white pixels become black; all black pixels become white.

```
void  FrameOval( Rect  *the_rect);

void  PaintOval( Rect  *the_rect);

void  FillOval( Rect    *the_rect,
                Pattern  the_pat );

void  EraseOval( Rect  *the_rect);

void  InvertOval( Rect  *the_rect);
```

Each of the previous five routines that perform operations on rectangles have an analogous Toolbox routine that performs the same operation on an oval. For each oval routine, the oval is drawn within the rectangle *the_rect*.

```
void  FrameRoundRect( Rect  *the_rect,
                      short  width,
                      short  height );

void  PaintRoundRect( Rect  *the_rect,
                      short  width,
                      short  height );

void  FillRoundRect( Rect    *the_rect,
                     short    width,
                     short    height,
                     Pattern  the_pat );

void  EraseRoundRect( Rect  *the_rect,
                      short  width,
                      short  height );

void  InvertRoundRect( Rect  *the_rect,
                       short  width,
                       short  height );
```

Each of the five routines that perform operations on rectangles have an analogous Toolbox routine that performs the same operation on a round rectangle. The amount of rounding to the corner of a round rectangle is determined by *width* and *height*.

```
void  FillCRect( Rect         *the_rect,
                 PixPatHandle  ppat_handle );
```

To fill a rectangle with a colored pattern, use *FillCRect()*. *FillCRect()* fills rectangle *the_rect* with the *PixPat* accessed through *ppat_handle*. The window that is being drawn to should be a color window created with a call to *GetNewCWindow()*.

```
void  FillCOval( Rect         *the_rect,
                 PixPatHandle  ppat_handle );
```

To fill an oval with a colored pattern, use *FillCOval()*. This routine fills the oval inscribed into *the_rect* with the *PixPat* accessed through *ppat_handle*. The window that is being drawn to should be a color window created with a call to *GetNewCWindow()*.

```
void  FillCRoundRect( Rect         *the_rect,
                      short         width,
                      short         height,
                      PixPatHandle  ppat_handle );
```

To fill a round rectangle with a colored pattern, use *FillCRoundRect()*. This routine fills the round rectangle described by *the_rect* with the *PixPat* accessed through *ppat_handle*. The window that is being drawn to should be a color window created with a call to *GetNewCWindow()*.

Events

This section describes the important Event Manager routines found in the Toolbox.

There are additional constants and data structures listed in Apple's *Events.h* header file. You *do not* have to *#include* this file in your projects. Both the THINK C *MacHeaders* and the Symantec C++

MacHeaders++ files include this header, and many others. By default, your THINK C and Symantec C++ projects contain *MacHeaders* or *MacHeaders++*.

Constants

```
#define    nullEvent       0
#define    mouseDown       1
#define    mouseUp         2
#define    keyDown         3
#define    keyUp           4
#define    autoKey         5
#define    updateEvt       6
#define    diskEvt         7
#define    activateEvt     8
#define    osEvt          15
```

After a call to *GetNextEvent()* or *WaitNextEvent()*, the *what* field of the returned *EventRecord* will contain one of the above constants.

```
#define    mDownMask               2
#define    mUpMask                 4
#define    keyDownMask             8
#define    keyUpMask              16
#define    autoKeyMask            32
#define    updateMask             64
#define    diskMask              128
#define    activMask             256
#define    highLevelEventMask   1024
#define    osMask             -32768
#define    everyEvent            -1
```

GetNextEvent() and *WaitNextEvent()* are passed a mask that tells them which events to watch for. Most applications will use *everyEvent* as this mask. The occurrance of any type of event will be reported to your program, and the logic of your program can then determine which event types to respond to.

```
#define    charCodeMask        0x000000FF
#define    keyCodeMask         0x0000FF00
#define    adbAddrMask         0x00FF0000
#define    osEvtMessageMask    0xFF000000
```

To determine which character is the result of a keystroke, perform an & operation on the *message* field of the most recent event and the *charCodeMask*.

Data Structures

```
struct EventRecord
{
   short  what;
   long   message;
   long   when;
   Point  where;
   short  modifiers;
};

typedef struct EventRecord EventRecord;
```

Unlike some record data structures, you'll access the fields of the *EventRecord* directly, without using a pointer or handle.

The *what* field holds the type of an event, such as *mouseDown* or *updateEvt*.

The *message* field holds information that varies from one event type to the next.

The *when* field gives the time on the system clock when the event occurred.

The *where* field holds the location of the cursor at the time the event occurred.

The *modifiers* field holds the modifier keys that were pressed at the time of the event. The Command and Option keys are examples of modifier keys.

Event Reporting

```
Boolean  GetNextEvent( short        event_mask,
                       EventRecord *the_event );
```

GetNextEvent() sets *the_event* to the next available event of the type or types specified by *event_mask*. To receive events of all types, set *event_mask* equal to the constant *everyEvent*. After *GetNextEvent()* receives the information that makes up *the_event*, it will remove it from the event queue in anticipation of handling the next event. *GetNextEvent()* will return a value of true if the event is of a type your program is looking for, as defined by *event_mask*. Otherwise it returns a value of false.

```
Boolean  WaitNextEvent( short          event_mask,
                        EventRecord   *the_event,
                        unsigned long  sleep,
                        RgnHandle      mouse_rgn );
```

The operation of *WaitNextEvent()* is similar to that of *GetNextEvent()*, described above. Additionally, *WaitNextEvent()* allows MultiFinder and System 7 software to switch control from the current program to another running application if there are no events in the event queue. This allows other applications to perform background tasks, even while yours application remains active and in the forefront.

The *sleep* parameter tells the system the maximum number of ticks that your program is willing to relinquish between events. A single tick is one sixtieth of a second. A *sleep* value of zero requests that the system return control to your program as soon as possible.

The *mouse_rgn* parameter is used to aid in cursor display. If your program changes the look of the cursor at different screen locations, you'll want to give *mouse_rgn* a value other than nil, or *0L*.

Mouse Reporting

```
void  GetMouse( Point  *mouse_loc );
```

GetMouse() returns the location of the mouse at the time the call is made. The location will be given in local coordinates—that is, *mouse_loc* will be described in terms of the coordinates of the current grafPort.

```
Boolean  Button( void );
```

Button() will return a value of true if the mouse button is down at the time of the call.

Windows

This section details the most commonly used routines that involve the Window Manager.

There are additional constants and data structures listed in Apple's *Windows.h* header file. You *do not* have to *#include* this file in your projects. Both the THINK C *MacHeaders* and the Symantec C++ *MacHeaders++* files include this header, and many others. By default, your THINK C and Symantec C++ projects contain *MacHeaders* or *MacHeaders++*.

Constants

```
#define    inDesk          0
#define    inMenuBar       1
#define    inSysWindow     2
#define    inContent       3
#define    inDrag          4
#define    inGrow          5
#define    inGoAway        6
#define    inZoomIn        7
#define    inZoomOut       8
```

FindWindow() returns the part of the window in which a mouse-down event occurred.

Data Structures

```
struct  WindowRecord
{
   GrafPort            port;
   short               windowKind;
   Boolean             visible;
   Boolean             hilited;
```

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

typedef  struct  WindowRecord  WindowRecord;
typedef  WindowRecord  *WindowPeek;

typedef  GrafPtr  WindowPtr;
```

You'll seldom have cause to directly access any of the fields of a *WindowRecord* other than the *port* member, the *GrafPort*. Instead, you'll indirectly access the fields using Toolbox calls. For the times you need direct access, use a pointer to the entire *WindowRecord*—a *WindowPeek*.

A *WindowPtr* points to the first field of the *WindowRecord* by the following definitions:

```
struct GrafPort
{
   [ GrafPort members ]
};

typedef  struct  GrafPort  GrafPort;
typedef  GrafPort  *GrafPtr;

typedef  GrafPtr  WindowPtr;
```

The above states that a *WindowPtr* is the same as a *GrafPtr*. A *GrafPtr* is a pointer to a *GrafPort*. A *WindowPtr* is a *GrafPtr*, and points to a *GrafPort*-the first member of the *WindowRecord* structure. See the QuickDraw section of this appendix for the complete definition of a *GrafPort*.

Window Allocation

```
WindowPtr  GetNewWindow( short      wind_ID,
                         Ptr        wind_storage,
                         WindowPtr  behind );
```

GetNewWindow() loads a window into memory using a 'WIND' resource. The description of the window is read in from the 'WIND' resource with ID *wind_ID*. Pass a nil pointer, *0L*, for the *wind_storage* if you want the Window Manager to choose the memory location for the window. A *behind* value of *(WindowPtr)-1L* places the window in front of all other windows, a value of nil, *0L*, places it behind.

```
WindowPtr  GetNewCWindow( short      wind_ID,
                          Ptr        wind_storage,
                          WindowPtr  behind );
```

GetNewCWindow() loads a color window into memory using a 'WIND' resource. If color attributes have been defined in the 'WIND' resource with ID *wind_ID*, they will appear in the window when it is displayed. The last two parameters are the same as for *GetNewWindow()*. Note that both *GetNewCWindow()* and *GetNewWindow()* return a *WindowPtr*. This pointer can be used in any Toolbox routines that require a *WindowPtr* as a parameter.

```
void  CloseWindow( WindowPtr  the_window );
```

CloseWindow() erases *the_window* and removes it from the list of open windows. It does not release the memory used by the window's *WindowRecord*. Use this routine only if you supplied the window storage in your call to *GetNewWindow()* or *GetNewCWindow()*. To free the memory associated with the *WindowRecord*, call *DisposPtr((Ptr)the_window)* after *CloseWindow()*.

```
void  DisposeWindow( WindowPtr  the_window );
```

DisposeWindow() erases *the_window* and removes it from the list of open windows. It also frees the memory used for *the_window's WindowRecord*. Use *DisposeWindow()* if you passed a nil pointer, *0L*, as the window storage in your call to *GetNewWindow()* or *GetNewCWindow()*.

Window Display

```
void  SetWTitle( WindowPtr  the_window,
                 Str255     title );
```

SetWTitle() sets the title of *the_window* to the text that makes up the *Str255* variable *title*.

```
void  GetWTitle( WindowPtr  the_window,
                 Str255     title );
```

GetWTitle() reads the current title of *the_window* and sets the *Str255* variable *title* to that value.

```
WindowPtr  FrontWindow( void );
```

FrontWindow() returns a *WindowPtr* to the active window that is, the window that is currently frontmost on the screen. If the screen is empty of windows, *FrontWindow()* will return a nil pointer—*0L*.

```
void  SelectWindow( WindowPtr  the_window );
```

SelectWindow() activates *the_window*. The previously active window is unhighlighted, *the_window* is placed in front of all others, *the_window* is properly highlighted, and an activate event is generated.

```
void  HideWindow( WindowPtr  the_window );
```

HideWindow() makes *the_window* invisible. It does not dispose of it. If *the_window* is already invisible, *HideWindow()* has no effect. If any other windows exist, the one that is behind *the_window* becomes the active window. To make the hidden window again visible, use *ShowWindow()*.

```
void  ShowWindow( WindowPtr  the_window );
```

ShowWindow() makes *the_window* visible. If *the_window* is already visible, *ShowWindow()* has no effect. *ShowWindow()* highlights *the_window* but does not change the front-to-back ordering of windows. To show a hidden window and bring it to the front, use *SelectWindow()* in conjunction with *ShowWindow()*. To make the shown window again hidden, use *HideWindow()*.

```
void  MoveWindow( WindowPtr  the_window,
                  short      horizontal,
                  short      vertical,
                  Boolean    front );
```

MoveWindow() moves *the_window* to the screen location specified by the second and third arguments. The top left corner of the window will be placed at the screen point defined by *horizontal* and *vertical*. The size of *the_window* will be unaffected. If the value of front is true, then *the_window* will become the active window.

```
void  DragWindow( WindowPtr  the_window,
                  Point      start_pt,
                  Rect      *drag_rect );
```

DragWindow() should be called in response to a *mouseDown* event in *the_window*'s drag region. The *start_pt* should be set to the location of the cursor when the mouse was pressed, as given in the *where* field of the *EventRecord*. Window movement will be restricted to the boundaries of the rectangle defined by *drag_rect*.

Windows and the Mouse

```
short  FindWindow( Point       the_point,
                   WindowPtr  *the_window );
```

A call to *FindWindow()* yields both the window (*the_window*) and the part of the window (the *short* return value) in which a *mouseDown* event occurred. The returned *short* value will be one of the constants listed above in the Constants section, such as *inDrag* or *inGrow*. Set *the_point* to the location of the cursor when the event occurred. This can be obtained from the *where* field of the *EventRecord*.

Updating

```
void  EraseRgn( RgnHandle  update_rgn );

void  BeginUpdate( WindowPtr  the_window );
```

Call *BeginUpdate()* in response to an *updateEvt* for *the_window*. After calling *BeginUpdate()*, call *EraseRgn()*, passing *EraseRgn()* the *visRgn* of *the_window*, as in: *EraseRgn(@the_window->visRgn)*. Then perform all the drawing necessary to draw the entire contents of the window. The *EraseRgn()* call will restrict the actual updating to only the area needed updating. After drawing to the window, call *EndUpdate()*.

```
void  EndUpdate( WindowPtr  the_window ):
```

EndUpdate() restores the *visRgn* of *the_window*. This region was altered during *BeginUpdate()*.

Dialogs

This section describes many of the Toolbox routines that involve the Dialog Manager.

There are additional constants and data structures listed in Apple's *Dialogs.h* header file. You *do not* have to *#include* this file in your projects. Both the THINK C *MacHeaders* and the Symantec C++ *MacHeaders++* files include this header, and many others. By default, your THINK C and Symantec C++ projects contain *MacHeaders* or *MacHeaders++*.

Data Structures

```
struct DialogRecord
{
   WindowRecord  window;
   Handle        items;
   TEHandle      textH;
   short         editField;
   short         editOpen;
   short         aDefItem;
};

typedef  struct  DialogRecord  DialogRecord;
typedef  DialogRecord  *DialogPeek;
```

```
typedef  WindowPtr  DialogPtr;
```

As with a *WindowRecord*, you'll seldom need direct access to any of the fields of a *DialogRecord*. You will instead use a *DialogPtr*. The first member of the *DialogRecord* is a *WindowRecord*. The first member of a *WindowRecord* is the *port*—the *GrafPort*. A *DialogPtr*, like a *WindowPtr*, points to a *GrafPort*. A *DialogPtr* can thus be used in Toolbox calls expecting a *WindowPtr* as an argument. See the Constants section of the Windows heading of this appendix for more information.

For the few times you need direct access to fields other than the port, use a pointer to the entire *DialogRecord*—a *DialogPeek*.

Dialog Allocation

```
DialogPtr  GetNewDialog( short      dlog_ID,
                         Ptr        dlog_storage,
                         WindowPtr  behind );
```

GetNewDialog() loads a dialog into memory using a 'DLOG' resource. The description of the dialog is read in from the 'DLOG' resource with ID *dlog_ID*. Pass a nil pointer, *0L*, for the *dlog_storage* if you want the Dialog Manager to choose the memory location for the dialog. A *behind* value of *(WindowPtr)-1L* places the dialog in front of all other windows, a value of nil, *0L*, places it behind.

There is no separate call to create a color dialog as there is for creating a color window. Instead, you use ResEdit to add color to any element—such as the frame or title bar—of the dialog's 'DLOG' resource. That will create a 'dctb' resource. Existence of the 'dctb' resource tells *GetNewDialog()* to base the new dialog on a color graphics port.

```
void  CloseDialog(DialogPtr the_dialog);
```

CloseDialog() erases *the_dialog* and removes its window from the list of open windows. It does not release the memory used by the dialog's *DialogRecord* or by the dialog's item list. Use this routine only if you supplied the dialog storage in your call to *GetNewDialog()*. To free the memory associated with the *DialogRecord*, call *DisposPtr((Ptr)the_dialog)* after *CloseDialog()*.

```
void  DisposDialog( DialogPtr  the_dialog );
```

DisposDialog() erases *the_dialog* and removes its window from the list of open windows. It also frees the memory used for *the_dialog's DialogRecord* and item list. Use *DisposDialog()* if you passed a nil pointer,*0L*, as the dialog storage in your call to *GetNewDialog()*.

Dialog Events

```
void  ModalDialog( ProcPtr  Filter_Function,
                   short   *the_item );
```

ModalDialog() performs event handling for a modal dialog box. When an event involves an enabled item, the item number of that item is returned to the program as *the_item*.

ModalDialog() optionally accepts a pointer to a filter function. If this value is nil, *0L*, *ModalDialog()* is responsible for all handling of the event. If a pointer to a filter function is included in the call, the filter function will handle some or all of the events. The filter function name, without parentheses, serves as the *ProcPtr*. The filter function is application defined. Its format is given below.

```
pascal  Boolean  Filter_Function( DialogPtr    the_dialog,
                                  EventRecord *the_event,
                                  short       *item );
```

Filter_Function() is an application-defined function that should be written to perform any dialog-related tasks not performed by *ModalDialog()*. The function can have any name, but it must have the three arguments listed. The first is a pointer to the active dialog. The *EventRecord* should be the event currently being handled. The item should be the *item* selected by the user.

```
Boolean  IsDialogEvent( EventRecord *the_event );
```

IsDialogEvent() determines if, at the time of the current event, the frontmost window was a dialog box. If a dialog box wasn't in the forefront the event is not dialog related, and *IsDialogEvent()* returns a value of false to the calling routine.

```
Boolean  DialogSelect( EventRecord *the_event,
                       DialogPtr   *the_dialog,
                       short       *the_item );
```

DialogSelect() does all the work for you if a dialog needs updating or activating. Call it after *IsDialogEvent()* has returned a value of true.

If the event was dialog related but wasn't an update or activate event, *DialogSelect()* doesn't handle it. Instead, *DialogSelect()* returns a pointer to the dialog and the item number of the clicked-on item for further processing by your program.

```
void  DlgCut( DialogPtr  the_dialog );
```

DlgCut() handles the Cut command for text within a dialog's edit text item.

```
void  DlgPaste( DialogPtr  the_dialog );
```

DlgPaste() handles the Cut command for text within a dialog's edit text item.

```
void  DlgCopy( DialogPtr  the_dialog );
```

DlgCopy() handles the Copy command for text within a dialog's edit text item.

```
void  DlgPaste( DialogPtr  the_dialog );
```

DlgPaste() handles the Paste command for text within a dialog's edit text item.

Alerts

```
short  Alert( short    alert_ID,
              ProcPtr  Filter_Function );
```

Alert() loads, displays, and handles and alert defined by an 'ALRT' resource with an ID of *alert_ID*. It displays no icon, as the other three forms of the *Alert()* function do. The *ProcPtr* argument is a pointer to an optional filter function that handles each event before processing by the

Alert() function. See *ModalDialog()* for more information on filter functions. *Alert()* returns a value of type *short* that contains the item number selected by the user.

```
short  StopAlert( short    alert_ID,
                  ProcPtr  Filter_Function );
```

StopAlert() is identical to *Alert()* except that it displays a stop-sign icon in the alert's top left corner.

```
short  NoteAlert( short    alert_ID,
                  ProcPtr  Filter_Function );
```

NoteAlert() is identical to *Alert()* except that it displays a message icon in the alert's top left corner.

```
short  CautionAlert( short    alert_ID,
                     ProcPtr  Filter_Function);
```

CautionAlert() is identical to *Alert()* except that it displays a cautionary icon in the alert's top left corner.

Dialog and Alert Items

```
void  ParamText( Str255  str_0,
                 Str255  str_1,
                 Str255  str_2,
                 Str255  str_3 );
```

ParamText() allows up to four strings to be substituted in an alert or dialog. If a static text item contains the string "^0", the text that comprises *str_0* will be substituted for "^0". In addition *str_1* will replace "^1", *str_2* will replace "^2", and *str_3* will replace "^3". Less than four strings can be defined in *ParamText()* by using one or more empty strings ("\p").

```
void  GetDItem( DialogPtr  the_dialog,
                short      the_item,
                short     *the_type,
                Handle    *the_handle,
                Rect      *the_rect );
```

To obtain information about a dialog item, pass *GetDItem()* a pointer to the dialog and the item number of the item in question. The item number for any item can be found in the dialog's 'DITL' resource. After *GetDItem()* has executed, *the_type* will contain the item's type, *the_handle* will hold a handle to the item, and *the_rect* will hold the display rectangle that holds the item.

```
void  SetDItem( DialogPtr  the_dialog,
                short      the_item,
                short      the_type,
                Handle     the_handle,
                Rect       *the_rect );
```

The description of an item can be changed using *SetDItem()*. All parameters are the same as they are for *GetDItem()*.

```
void  GetIText( Handle  the_item,
                Str255  the_str );
```

GetIText() returns the text from a text item in a dialog. Parameter *the_item* is a handle to the item. This handle can be obtained by first calling *GetDItem()*. After the call to *GetIText()*, *the_str* will hold the contents of the text item.

```
void  SetIText(Handle  the_item,
               Str255  the_str );
```

SetIText() changes the text in a dialog text item. Parameter *the_item* is a handle to the item and can be obtained by first calling *GetDItem()*. The *Str255* parameter *the_str* is the text to set the item to.

```
void  SetCTitle( ControlHandle  the_control,
                 Str255         title );
```

SetCTitle() sets the title of *the_control* to the text in *title*. You can get a *Handle* to *the_control* by first calling *GetDItem()*. The returned *Handle* should be typecast to the proper type when calling *SetCTitle()*. Assuming *the_handle* is of type *Handle* and was returned by *GetDItem()*, a call to *SetCTitle()* would look like the following: *SetCTitle((ControlHandle)the_handle, title);*

Use *SetCTitle()* to change the title of a check box or radio button.

```
void  GetCTitle( ControlHandle  the_control,
                 Str255         title );
```

GetCTitle() returns the current title of the item pointed to by *the_control*. See *SetCTitle()* for information on obtaining this *ControlHandle*.

```
void  SetCtlValue( ControlHandle  the_control,
                   short          the_value );
```

SetCtlValue() sets the value of the item pointed to by *the_control*. See *SetCTitle()* for information on obtaining this *ControlHandle*. Parameter *the_value* should be either a 1 or 0. A value of 1 turns the control on; a value of 0 turns it off.

Use *SetCtlValue()* to change the value, or state, of a check box or radio button.

```
short  GetCtlValue( ControlHandle  the_control );
```

GetCtlValue() returns the value of the item pointed to by *the_control*. See *SetCTitle()* for information on obtaining this *ControlHandle*. The returned *short* type will be either 1 or 0. A value of 1 means the control is on; a value of 0 means that it is off.

Use *SetCtlValue()* to change the value, or state, of a check box or radio button.

Menus

This section describes the important Menu Manager routines found in the Toolbox.

There are additional constants and data structures listed in Apple's *Menus.h* header file. You *do not* have to *#include* this file in your projects. Both the THINK C *MacHeaders* and the Symantec C++ *MacHeaders++* files include this header, and many others. By default,

your THINK C and Symantec C++ projects contain *MacHeaders* or *MacHeaders++*.

Constants

```
#define    normal          0
#define    bold            1
#define    italic          2
#define    underline       4
#define    outline         8
#define    shadow       0x10
#define    condense     0x20
#define    extend       0x40
```

The style of the text of a menu item can be changed with a call to *SetItemStyle()*. Pass in one or a combination of the above *Style* constants.

Data Structures

```
typedef  unsigned  char  Style;
```

A call to *SetItemStyle()* changes the text style of a menu item. Use the *Style* constants defined above. To combine *Styles*, declare a variable of type *Style*, then add the constants that will yield the desired combination:

```
Style  item_style;
item_style = bold + italic + shadow;

struct MenuInfo
{
   short    menuID;
   short    menuWidth;
   short    menuHeight;
   Handle   menuProc;
   long     enableFlags;
   Str255   menuData;
};
```

```
typedef  struct  MenuInfo  MenuInfo;
typedef  MenuInfo  *MenuPtr,  **MenuHandle;
```

As with a *WindowRecords* and *DialogRecords*, you'll seldom need direct access to any of the fields of a *MenuInfo*. You will instead use a *MenuHandle*.

Menu Allocation and Display

```
Handle  GetNewMBar( short  mbar_ID );
```

GetNewMBar() creates a menu list, using the individual 'MENU's specified in the 'MBAR' resource with an ID of *mbar_ID*. The list contains a handle to each individual menu that will appear in the menu bar. *GetNewMBar()* does not install the individual menus or display the menu bar.

```
void  SetMenuBar( Handle  menu_list );
```

SetMenuBar() installs the individual menus in the menu bar specified by *menu_list*. This handle should be the one returned by *GetNewMBar()*. The effect of *SetMenuBar()* is to make *menu_list* the current menu list; a resource file can have more than one 'MBAR' resource.

```
MenuHandle  GetMHandle( short  menu_ID );
```

GetMHandle() returns a handle to the 'MENU' with a resource ID of *menu_ID*. You'll then be able to change characteristics of this menu and items in it using other Toolbox routines.

```
void  AddResMenu( MenuHandle  the_menu,
                  ResType     the_type );
```

AddResMenu() locates all items of type *the_type* and appends them to *the_menu*. For the Apple menu, *the_type* should be 'DRVR'. *AddResMenu()* adds all the desk accessories in the user's system to the Apple menu. Under System 7, *AddResMenu()* will also append all items located in the Apple Menu Items folder in the System Folder. The *MenuHandle the_menu* should be obtained with a call to *GetMHandle()*.

```
void  DrawMenuBar( void );
```

None of the preceding calls actually displays the menu bar on the screen. After a menu setup has been performed, call *DrawMenuBar()* to draw it.

Menu Selections

```
long  MenuSelect( Point start_pt );
```

When an event is of *mouseDown* type, and it is further determined that the location of the mouse down was *inMenuBar*, call *MenuSelect()*. Pass the *where* field of the event as the *start_pt*. *MenuSelect()* handles the dropping and displaying of menus as the user moves the mouse over the menu bar. Both the user-selected menu and menu item will be determined by *MenuSelect()* and saved in the returned *long* type.

```
long  MenuKey( short chr );
```

If a *keyDown* event occurs, and the Command key was pressed simultaneously, call *MenuKey()*. Given the typed character *chr*, *MenuKey()* will determine which menu and menu item this keystroke combination is equivalent to, and return it in the *long* type. The value returned by *MenuKey()* will be identical to that which *MenuSelect()* would return if the menu choice had been made with the mouse rather than with a Command-key equivalent.

Hierarchical Menus

```
MenuHandle  GetMenu( short  resource_ID );
```

When *GetNewMBar()* reads in the 'MENU' descriptions of the menu that will appear in the menu bar, it takes note of submenu IDs but does not read in their descriptions. *GetMenu()* does this. The *resource_ID* is the ID of the 'MENU' that represents the submenu of the hierarchical menu. Call *InsertMenu()* after calling *GetMenu()*.

```
void  InsertMenu( MenuHandle  the_menu,
                  short       before_ID );
```

After reading in the description of a submenu using *GetMenu()*, call *InsertMenu()* to insert the submenu into the menu list. The parameter

the_menu should be the *MenuHandle* returned by *GetMenu()*. Assign *before_ID* a value of -1 to let the Menu Manager know this is a submenu rather than a menu in the menu bar.

Changing Menu Characteristics

```
void  SetItem( MenuHandle  the_menu,
               short       the_item,
               Str255      the_str );
```

SetItem() changes the text of menu item *the_item* in *the_menu*. The new text that will appear in the menu will be that of *the_str*. Use *GetMHandle()* to get a handle to the menu.

```
void  GetItem( MenuHandle  the_menu,
               short       the_item,
               Str255      the_str );
```

GetItem() gets the text of menu item *the_item* in *the_menu* and places it in the *Str255* variable *the_str*. Use *GetMHandle()* to get a handle to the menu.

```
void  DisableItem( MenuHandle  the_menu,
                   short       the_item );
```

DisableItem() disables the menu item *the_item* in *the_menu* by dimming it and ignoring user attempts to select it. If *the_item* is given a value of zero, the entire menu will be disabled. The menu name in the menu bar, and all menu items in the menu, will become dim. Use *GetMHandle()* to get a handle to the menu. Use *EnableItem()* to enable a disabled menu or menu item.

```
void  EnableItem( MenuHandle  the_menu,
                  short       the_item );
```

EnableItem() enables the menu item *the_item* in *the_menu* by highlighting the dimmed item. If *the_item* is given a value of zero, the entire menu will be enabled. The menu name in the menu bar and all menu items in the menu will be highlighted. Use *GetMHandle()* to get

a handle to the menu. Use *DisableItem()* to disable an enabled menu or menu item.

```
void  CheckItem( MenuHandle  the_menu,
                 short       the_item,
                 Boolean     checked );
```

CheckItem() places a checkmark to the left of the text in *the_item* in *the_menu*, if *checked* is true. If *checked* is false, the checkmark will be removed from the left of that item. Attempting to check an already checked item has no effect. The same is true for an attempt to uncheck a menu item that has no checkmark by it. Use *DisableItem()* to disable an enabled menu or menu item. Use *GetMHandle()* to get a handle to the menu.

```
void  SetItemStyle( MenuHandle  the_menu,
                    short       the_item,
                    Style       chk_style );
```

The text of a menu item does not have to appear in its default style of *plain*. *SetItemStyle()* changes the style of the text of *the_item* in *the_menu* to that given by *chk_style*. The style can be one or any combination of *Styles* from the set listed in the Constants heading of this section. Use *GetMHandle()* to get a handle to the menu.

```
void  GetItemStyle( MenuHandle  the_menu,
                    short       the_item,
                    Style      *chk_style );
```

GetItemStyle() returns the *Style* of the text in *the_item* in *the_menu*. Use *GetMHandle()* to get a handle to the menu.

Memory

This section describes the important Toolbox routines that work with memory.

There are additional constants and data structures listed in Apple's Memory.h header file. You do not have to #include this file in your pro-

jects. Both the THINK C MacHeaders and the Symantec C++ MacHeaders++ files include this header, and many others. By default, your THINK C and Symantec C++ projects contain MacHeaders or MacHeaders++.

Memory Allocation

```
void  MaxApplZone( void );
```

At program startup the application's heap is set to a small size. If left in that state, it will grow as objects are loaded into it. For more efficient heap management, call *MaxApplZone()* at program startup to immediately increase the heap to its maximum size.

```
void  MoreMasters( void );
```

Master pointers are allocated in blocks. When your program starts up, the Memory Manager gives you one block. If, during the course of program execution, your program runs out of master pointers, the Memory Manager will place another block in memory. This can lead to fragmentation. Call *MoreMasters()* four or five times at the very start of your program to ensure that the Memory Manager doesn't do so later on.

```
Handle  NewHandle( Size  num_bytes );
```

NewHandle() returns a handle to a relocatable block of memory. The size of the block is *num_bytes* bytes.

```
void  DisposHandle( Handle  the_handle );
```

DisposHandle() frees the memory occuppied by the block accessed by *the_handle*. Once disposed of, any other existing handles that access this same block become invalid.

```
Ptr  NewPtr( Size  num_bytes );
```

NewPtr() returns a pointer to a nonrelocatable block of memory. The size of the block is *num_bytes* bytes.

```
void  DisposPtr( Ptr  the_ptr );
```

DisposPtr() frees the memory occupied by the block accessed by *the_ptr*. Any other existing pointers that point to this same block become invalid Once the memory is disposed of.

```
void  ExitToShell( void );
```

Always check the result of a memory allocation. If the allocation fails, it will return a value of nil. To avoid a crash, call *ExitToShell()* at that point. A call to *ExitToShell()* prevents a frozen screen and allows your application to exit gracefully by releasing the application heap and returning the user to the Finder.

```
void  UnloadSeg( ProcPtr  Routine_Name );
```

Macintosh programs are segmented. Segments will be loaded and unloaded from memory if the memory allotted to an application becomes low. To mark a segment as purgeable, call *UnloadSeg()*. *Routine_Name* is the name of one of the routines in the segment. It acts as a pointer to the routine and tells *UnloadSeg()* which segment to mark as purgeable. *UnloadSeg()* does not actually unload the segment; it just gives the system permission to do so if memory becomes restricted.

Utilities

This section describes the important general-purpose functions found in the Toolbox.

There are additional constants and data structures listed in Apple's Resources.h and ToolUtils.h header files. You do not have to #include these files in your projects. Both the THINK C MacHeaders and the Symantec C++ MacHeaders++ files include these headers, and many others. By default, your THINK C and Symantec C++ projects contain MacHeaders or MacHeaders++.

Constants

```
#define    curSysEnvVers     2
```

The *SysEnvirons()* routine returns information about the system of the machine on which your program is running. Use the constant *curSysEnvVers* in calls to *SysEnvirons()*. Should Apple update the *SysEnvirons()* over time, the *curSysEnvVers* value will be changed and your calls can remain unchanged.

```
#define    envMac            -1
#define    envXL             -2
#define    envMachUnknown     0
#define    env512KE           1
#define    envMacPlus         2
#define    envSE              3
#define    envMacII           4
#define    envMacIIx          5
#define    envMacIIcx         6
#define    envSE30            7
#define    envPortable        8
#define    envMacIIci         9
#define    envMacIIfx        11
```

SysEnvirons() fills the fields of a *SysEnvRec*. Those fields are given below in the *SysEnvRec* structure listing. You may find the *machineType* field the most important. You can check the value of that field at program start up. If the returned value indicates that your program is running on a machine that is too old (as determined by you), you may wish to exit the program. Information from all of the other fields can be better obtained by a call to *Gestalt()*, which is described later in this section and throughout this book.

```
#define    iBeamCursor    1
#define    crossCursor    2
#define    plusCursor     3
#define    watchCursor    4
```

The standard arrow-shaped cursor can be changed to any one of four system-defined cursors using calls to *GetCursor()* and *SetCursor()*. Use one of the above constants in the call to *GetCursor()*.

Data Structures

```
struct SysEnvRec
{
   short    environsVersion;
   short    machineType;
   short    systemVersion;
   short    processor;
   Boolean  hasFPU;
   Boolean  hasColorQD;
   short    keyBoardType;
   short    atDrvrVersNum;
   short    sysVRefNum;
};

typedef struct SysEnvRec SysEnvRec;
```

A call to *SysEnvirons()* fills a *SysEnvRec* with system information about the Macintosh on which your program is currently running. In most cases, you'll want to use the newer *Gestalt()* Toolbox function, which provides more information. On Macintoshes running older system software, however, *Gestalt()* may not be available.

```
struct Cursor
{
   Bits16 data;
   Bits16 mask;
   Point hotSpot;
};

typedef  struct  Cursor  Cursor;
typedef  Cursor  *CursPtr,  **CursHandle;
```

The system defines five cursors. You won't have to access fields of the *Cursor* structure itself. Instead, you use *GetCursor()* to receive a *CursHandle* with which to work.

System Features

```
OSErr  SysEnvirons( short      version,
                    SysEnvRec *sys_env_rec );
```

A call to *SysEnvirons()* fills the *SysEnvRec sys_env_rec* with system information about the machine currently running your program. Set *version* equal to the constant *curSysEnvVers*. You can then examine fields of the *SysEnvRec*. The *SysEnvRec* structure is given under the Data Structures heading of this section.

```
OSErr  Gestalt( OSType  selector,
                long   *response );


long  NGetTrapAddress( short     trap_num,
                       TrapType  trap_type );
```

When passed trap number *trap_num* and the type of trap, *trap_type*, *NGetTrapAddress()* returns the address of the trap, or routine. To test for the availability of a Toolbox routine, call *NGetTrapAddress()* twice. On the first call, set *trap_num* to the trap number of the routine in question. On the second call, set *trap_num* to the unimplemented trap number. If the returned results of both calls *are not* equal, the trap exists and it is safe to call that routine.

Extracting Information From Long Ints

```
short  HiWord( long long_num );
HiWord() returns the high-order 16 bits of the 32-bit long_num.

short  LoWord( long long_num );
```

LoWord() returns the low-order 16 bits of the 32-bit *long_num*.

Causing a Delay

```
void  Delay( long  num_ticks,
             long *final_ticks );
```

Delay() pauses your program for *num_ticks* ticks. A single tick is one sixtieth of a second. When the pause is completed, *final_ticks* will be filled in with the number of ticks from system startup to the end of the delay.

Don't attempt to use a loop, as in:

```
for ( i=0; i<10000; i++ )
   ;  /* do nothing, just killing time */
```

Rather, use the *Delay()* routine. A loop is processor dependent; That is, a loop will execute more quickly on a faster processor. The *Delay()* routine is processor independent its delay effect is the same on all CPUs.

Cursors

```
CursHandle  GetCursor( short  cursor_ID );
```

GetCursor() loads the 'CURS' resource specified by *cursor_ID* into memory and returns a *CursHandle* to it. It does not display the cursor. Use *SetCursor()* for that.

```
void  SetCursor( Cursor  *cursor_handle );
```

SetCursor() changes the shape of the cursor to that specified by the cursor. First call *GetCursor()* to get *cursor_handle*. Dereference that handle once to get a pointer to a cursor, as required by *SetCursor()*.

```
void  InitCursor( void );
```

InitCursor() sets the cursor to the familiar arrow shape. You do not have to call *GetCursor()* first.

Loading Resources

```
Handle  GetResource( ResType  the_type,
                     short    the_ID );
```

GetResource() returns a generic handle to the resource with a resource ID of *the_ID*. The parameter *the_type* can be any resource type. Include single quotes around the type, as in this call that loads a sound resource with an ID of 9000:

```
GetResource( 'snd ', 9000 );
```

Sound

This section describes the Sound Manager routines covered in Chapter 3 of this book.

There are additional constants and data structures listed in Apple's *Sound.h* header file. You must *#include* this file in your projects if you're going to use 'snd ' resources in them. The *sound.h* file is not used as often as many other headers, so it has not been included in either the THINK C MacHeaders or the Symantec C++ MacHeaders++ files.

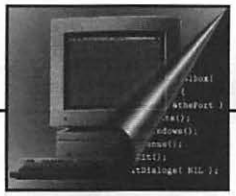
```
#include <Sound.h>
```

Playing a Sound

```
OSErr  SndPlay( SndChannelPtr  the_channel,
                Handle         sound_handle,
                Boolean        async );
```

SndPlay() plays a 'snd ' resource that has been loaded into memory. First call *GetResource()* using 'snd ' as the first parameter and the resource ID of the 'snd ' as the second parameter. *GetResource()* will return a handle to the sound; use this as *sound_handle*. Pass a value of true for async if this is the only sound that will be playing (asynchronous) or false if there will be multiple sounds playing at the same time (synchronous). See the Utilities section of this appendix for information on *GetResource()*.

Note that your 'snd ' resource should have a resource ID greater than 8192 so that it won't conflict with Apple's reserved 'snd ' resource numbering 0 to 8191.



Index

Symbols

#define directive 41
#include directive 40–41
& bit operator 305, 540–541
$ trap numbering 376
0x0 hexadecimal format 539–541
32-bit clean 450–452, 475

A

AddResMenu function 294–295
Alert function 227
alerts
    cancelling 227
    returned value 227
    variations 228
    defined 224
    resources 224–226
    item types 225–226
    item numbers 226
All_Done variable 164, 219
animation
    using 'PICT's 86–90
Apple Menu Items folder 294–295
application frameworks 14
application partition
    defined 50
    A5 World 52
    stack 52, 58
    heap 54, 58
    size 65, 67–68, 471–481
applications
    internationalizing 77
    localizing 77
    Type identifier 96
    Creator identifier 96, 102
    APPL type 96
    Signature 97
    quitting 164, 219
    terminating abruptly
        See ExitToShell function
    building 452–454
    size, setting 471–481

B

Bedrock 14
BeginUpdate function 179, 193
BIOS 28
bit-mapped graphics
    versus text-based 16
buttons
    drawing 215

C

casting *See* typecasting
CautionAlert function 228
central processing unit type 386
CGrafPort data type 136
character 433
charCodeMask constant 329
check box item 239–240
CheckItem function 315
checkmarks 315–318, 361–362
clip art 81
CloseWindow function 445
color
    dialog boxes 258–259
    displays *See* pixels, depth
    windows 136
Color QuickDraw 134–136, 385, 394
compaction 60–61
compressed files 91
computer characteristics
    See machine features
control item 239
ControlHandle data type 240
coordinate system
    See pixels, coordinate pair
CPU type 386
Creator 96, 102
CurHandle data type 143
cursors
    system 143
    using 143–144
curSysEnvVers constant 378

D

debugger 258
#define directive 41
definition functions 452
Delay function 90
desk accessories
    ports 123
    program interaction 167
    menu location 291, 294
    drivers 294–295
    storing 294
desktop
    rebuilding 102–103
    objects 295
    See also Finder
device
    defined 393
    list 393
dialog boxes
    item types 225–226, 229–230
    windows and 260
    item numbers 226
    defined 228
    modal 228, 241, 244–246
    modeless 228
    items, adding 231
    pictures in 232–236
    icons in 232, 234–236
    user items 236, 250
    displaying 236
    visibility 244
    events in 247–250
    color 258–259
    text in 323–331, 359–361
    filter function 325–331
    enabled items 326
dialog items
    mouse clicks in 233, 244
    picture size 233
    information about 237
    handle to 237

edit text 238, 329–330
check box 239–240
value 239
control item 239
radio button 240–241, 277
multiple use of 262
Dialog Manager 251–252
DialogPtr data type 242
DialogRecord data type 242
DialogSelect function 249
DisableItem function 312–314
disk access 460
DisposDialog function 246
DisposeWindow function 173, 219–220, 445
DisposPtr function 219, 445
DisposHandle function 294
DlgCopy function 330
DlgCut function 330
DlgPaste function 330
Do_User_Item function 253–257
drag region 390–392
Drag_Rect variable 172
DragWindow function 171
Draw_Moving_Picture function 88–90, 113
drawing
shapes 125–130
rectangles 126–128
ovals 128–129
round rectangles 129–130
color 140–141
DrawMenuBar function 295
DrawPicture function 85, 88
DrawString function 45
See also strings
driver 294–295

E

EnableItem function 312–314
EndUpdate function 179, 193
EraseOval function 128
EraseRect function 128
EraseRgn function 179
EraseRoundRect function 130
error handling 75–77, 481–485
event
types 165–167
defined 18
loop 19, 45
processing 19
queue 19
mouse down 45
handling 164–169
window-related 170–176
activate 177, 190–191
event-driven programming 17–22
event record
See EventRecord data type
EventRecord data type
where field 18, 217–218
what field 19
message field 178, 191, 305
modifier field 305, 329
example source code
decompressing 5–8
exiting a program
See application, quitting
ExitToShell function 76, 113, 482

F

file naming convention 36
FillOval function 128
FillRect function 126
FillRoundRect function 130
filter function 325–331
Finder
defined 34
exiting to *See ExitToShell* function
displaying icons in 96
FindWindow function 248
floating point unit type 387–388

font
size 431
type 431
See also strings
FPU type 387–388
fragmentation 58–59, 439–446
FrameOval function 128
FrameRect function 126
FrameRoundRect function 130
FrontWindow function 174
functions
prototypes 41
Macintosh format 46
naming convention 46
See also individual function names

G

GDevice data type 393, 398
GDeviceList function 398, 402
GDHandle data type 393, 398
Gestalt function 135, 324–325, 379–390
GestaltEqu.h header file 135, 383
Get_Min_Pixel_Depth function 402
Get_Pixel_Depth function 399, 402
Get_Some_Strings function 112
GetCtlValue function 239–240
GetCursor function 143
GetDItem function 237, 253
GetIndString function 78, 112, 320
GetItem function 300, 320
GetItemStyle function 322
GetIText function 238
GetMainDevice function 393
GetMessage function
GetMHandle function 294–295, 309
GetNewCWindow function 136, 142, 258
GetNewDialog function 236, 244, 246, 259
GetNewMBar function 293, 309
GetNewWindow function 28, 43–44, 160–161
GetNextDevice function 402
GetNextEvent function 19–22, 163
GetPattern function 133
GetPenState function 123
GetPicture function 83, 85
GetPixPat function 138
GetPort function 122, 178
GetResource function 94–95, 113
global variables 42
See also individual variables
GlobalToLocal function 218
GrafPort
defined 118
window relationship 162
See also graphics ports
GrafPtr
defined 118
graphical user interface
programming challenges 15
graphics
bit-mapped 16
graphics pen
defined 119
moving 119
size of 120
graphics ports
defined 118
changing 121–123
characteristics of 123–125
color 136
gray region 391–392
GrayRgn global variable 391–392

H

Handle_Apple_Choice function 300
Handle_Dialog_Event function 247
Handle_Keystroke function 304
Handle_Menu_Choice function 298
Handle_Modal_Dialog function 244

Handle_Mouse_Down function 46, 173, 297
Handle_One_Event function 165–166, 246
Handle_Update function 177–182
Handle data type 240
handles
  defined 63
  use of 63–64
  dereferencing 144–145
header files
  MacHeaders 40
  example of 525–527
hexadecimal format 539–541
HideWindow function 173
hierarchical menus 305–311
HiWord function 299
HLock function 144
Holt, Joe 480
HUnlock function 144
HyperCard 14

I

icons
  application 95–103
  color 99–101
  monochrome 99–100
  see also resource, types
In Action! software
  conjunction with book 12
  decompressing 5–8
  purpose of 10
  running animations 11
  selecting topics 12
  using 9–12
#include directive 40–41
InitCursor function 143
InitGraf function
Initialize_Toolbox function 116
InsertMenu function 309
InvertOval function 128
InvertRect function 128
InvertRoundRect function 130
IsDialogEvent function 248–249

K

keystrokes
  handling of 304–305, 329

L

Line function 16, 120
LineTo function 120
long data type 298
LoWord function 299

M

MacApp 14
MacHeaders 40
machine features
  CPU type 386
  FPU type 387–388
  operating system 389–390
  QuickDraw version 383, 385
  RAM amount 386–387
  response parameter 382
  selector codes 382
  type 381, 388–389
Macintosh User Interface Toolbox
  See Toolbox functions
main function 42
main screen 393
managers
  defined 32
  See also individual managers
master pointers
  allocating 446–449
  blocks 446–447
  defined 61
  use of 61–64
MaxApplZone function 449

memory
application partition
See application partition
attributes 55, 438
compaction 60–61
error messages 67
fragmentation 58–59, 439–446
locked blocks 438
locking 144–145
master pointers 446–449
nonrelocatable blocks 61, 438, 4
40–445
objects 438
overflow 65
partitions 49
purgeable blocks 439
relocatable blocks 61, 438
reserving 578
system partition
See system partition
unlocked blocks 438
unpurgeable blocks 439
Memory Manager 55, 438
menu bar
adding menus to 291–292
order of menus in 292
parts of 278–288
setting up 293–296, 353–354
Menu Manager 293
MenuHandle function 294
MenuKey function 305
menus
About item 290
characteristics 311–313
checkmarks 315–318, 361–362
desk accessories in 291, 294, 300
disabling 288, 311–314
enabling 288, 311–314
extracting item number 298
flashing 297
handles to 295
hierarchical 305–311
highlighting 297
item length 319
items 288
keyboard equivalent 288, 302–303
list 294
MDEF ID 307–308
mouse click in 296–301, 356–359
resource ID 292, 306–307
resources 289–292
selections 297
separator lines 288, 290
styles in 320–323
text changes 318–320
tracking mouse in 297, 301
MenuSelect function 297, 314
modal dialog box 228, 241, 244–246
ModalDialog function 245–246,
326–331
modeless dialog box 228
monitors
center point 392–395
color 396–406
displaying color pictures 397
drag region 390–392
gray region 391–392
main screen 393
multiple 390–395, 400–401
pixel depth *See* pixels, depth
MoreMasters function 447–449
mouse events
close box 173–174
desk accessory 175–176
drag bar 171–172
handling 167–169
menu bar 175–176
window content 174–175
mouseDown event type 296
Move function 119
MoveTo function 16, 44, 119
MoveWindow function 183

MultiFinder 65–66
Multifinder_Present variable 166
MyWindPeek structure 187

N

NewHandle function 446
NewPtr function 189, 246, 440
NGetTrapAddress function 375–378
nil pointer 43, 161, 189
noErr constant 95, 383
nonrelocatable blocks 61, 438, 440–445
NoteAlert function 228

O

OpenDeskAcc function 300
operating system
    ROM-based 31
    routines 31
    Toolbox comparison 31
OSTrap data type 376

P

PaintOval function 128
PaintRect function 127
PaintRoundRect function 130
ParamText function 335–337, 363, 484–485
part codes 168
pascal keyword 254
patches 33
PatHandle data type 133
Pattern data type 126, 133
patterns
    changing 127
    color 137–140
    drawing lines with 127
    filling shapes with 126
    standard 126, 131
PenPat function 127, 139
PenPixPat function 139
PenSize function 120
PenState data type 123
performance degradation 460
picFrame struct member 83
PicHandle data type 83, 85, 88
Picture data type 83
pictures
    clip art 81
    creating 81, 86–87, 180–181
    displaying 83–86
    drawing 81
    painting 81
    resource ID 88
    saved as resources 81–82
    size of 443
pixels
    bits per pixel 396–397
    coordinate pair 16, 117, 218
    defined 16
    depth 396–398, 405
PixMap data type 398
PixMapHandle data type 398
PixPat data type 137
PixPatHandle data type 138
Play_A_Sound function 113
Point data type 218
pointers
    generic 189
    integers as 44
    memory size of 43
    nil 43, 161, 189
    similarities 243
    See also master pointers
programs *See* applications
PtInRect function 218
purgeable blocks 439

Q

queue 19
QuickDraw
    color *See* Color QuickDraw
    defined 115

initializing 116
version 383, 385
quitting a program
See application, quitting

R

radio button item 240–241, 277
Rect data type
defined 83–85
pointer to 84
rectangle
basis of other shapes 125
coordinates 84
mouse clicks in 218
relocatable blocks 61, 438
ResEdit resource editor
creating resource file 37
creating resources 23
defined 23
editing with 26
Resourcerer resource editor 74
resources
creating 23, 37
compiling 25
defined 22
editing *See* ResEdit resource editor
errors 75–76
file versus program 452–454
IDs 44
importance of 74
merging 453–454
names 23
ResEdit *See* ResEdit resource editor
types
'ALRT' 75, 224–226
'BNDL' 75, 95–101, 488
'CODE' 25, 74, 455
'CURS' 143
'dctb' 259
'DITL' 24, 75, 24–226, 230–236
'DLOG' 24, 75, 230
'DRVR' 75, 294
'FREF' 101
'ICN#' 75, 99, 101
'icl4' 99, 101
'icl8' 99, 101
'ICON' 232
'ics#' 99
'ics4' 99
'ics8' 99
'MBAR' 24, 289, 291–292
'MENU' 24, 289–291, 303, 306
'PAT ' 131–132
'PICT' 75, 81–90
'ppat' 137
'SIZE' 75, 473–477
'snd ' 91–95
'STR#' 75, 77–80, 112
'wctb' 141
'WIND' 24, 28, 75, 160
source code and 28
using 133
response parameter 382

S

scanf function 21
screen
as a port 119
screenBits.bounds 172
segmentation
defined 455
loading 462
attributes 462–465
unloading 467–471
'CODE' 455, 461–464
source files 455, 458–461
preloaded 461
size barrier 459–460
naming 457
examples of 469, 497–498
main segment 457, 461–467
segment 0 457

selector codes 382
SelectWindow function 174
separator lines 288, 290
Set_Window_Drag_Boundaries function 172, 391
SetCtlValue function 239–240
SetCursor function 143
SetDItem function 253
SetItem function 318–320
SetItemStyle function 320–322
SetIText function 238
SetMenuBar function 294
SetPenState function 123
SetPort function 44, 79, 118, 122, 178
SetRect function 84, 125
SetWTitle function 80, 112
ShowWindow function 244
Signature 97
sizeof function 443
SndPlay function 94–95, 113
Sound Manager 94
sound
  creating 91
  files 92
  obtaining 91
  memory size of 95
  playing 94–95, 113
  resources 92–93
  storing 91
  transferring 92
Sound.h header file 94
source code
  multiple files 456, 495
StopAlert function 228
Str255 data type 78, 238, 542
strings
  converting to numbers 238
  displaying 112
  format of 45
  length 542
  memory content 542
  retrieving from 'STR#' 78, 491–494
  using 79–80
  See also Str255 data type
StringToNum function 238
StuffIt program 91
Style data type 320–321
SuperCard 14
Swatch application 480–481
Symantec C++ compiler 35
SysBeep function 46
SysEnvirons function 378–379
SysEnvRec data type 378–379
System 7
  checking for 324–325
System file
  defined 33
  patches in 33
system global variables *See individual variables*
system partition
  contents of 50
  defined 50
  global variables 51
  heap 51
system software
  defined 32
system version
  checking for 324–325
SystemClick function 175
SystemTask function 166

T

TextFont function 431
TextSize function 431
THINK C compiler
  Creator setting 102
  file naming convention 36
  header files 40
  multiple source files 456, 495–498
  program examples 35
  project file 36

resource merging 25
segmentation 455–465
size of applications 473
Toolbox functions
availability of 135
correct usage of 29
defined 27
initialization of 42
naming convention 30
number of 29
presence of 375–378
RAM routines 369–374
ROM routines 27, 369–374
traps *See* traps
ToolTrap data type 376
TrackGoAway function 173
traps
C language definitions 375
defined 368
dispatch table 370–374
numbers 368, 376–377
RAM 368–374
unimplemented 371–374
Traps.h header file 164, 375
typecasting
ControlHandle 240
WindowPtr 161, 186, 189

U

UnloadSeg function 467–471, 532–533
unlocked blocks 438
unpurgeable blocks 439
user items
definition routine 254–257
handling 257, 278
multiple 256–257
purpose 250–251
resources 251
setting up 253
text in 279
updating 251–255

W

WaitNextEvent function 166
Window Manager 32, 177
WindowPeek data type 162, 185–190
WindowPtr data type 162, 185–190
WindowRecord data type 118, 161–162, 185–190
windows
activating 177
closing 219–220, 435
color 136
creating 160–161
dialogs and 260
drawing to 44
dynamic data in 181
events in 170–176
expanding record 185–190
frontmost 174
keeping track of 212
memory size 211
moving 183
multiple 184–193
snapshot 181
storage 188, 445
title 80, 184, 211–212
types 187–190, 209–211
updating 177
visibility 183–184
Windows.h header file 40
WinMain function 42

About This Disk

The In Action! Mac Techniques software is a Macintosh program created specifically for the book *Macintosh Programming Techniques.* Through the use of over two dozen animations, this software tutorial supplements and enhances the understanding of the concepts presented in this book.

The one 1.4 Mb disk contains one self-extracting compressed file. When expanded, this file gives you the In Action! Mac Techniques program and the THINK C source code for the nine example programs listed in this book. Self-extracting means that you need no additional software to decompress these files—everything you need is right here on this disk.

This disk is a Macintosh 1.4 Mb high-density disk. All newer model Macintosh computers come with the SuperDrive—a 1.4 Mb high-density floppy drive. If you have an older Macintosh with an 800 K double-density floppy drive, you won't be able to use this disk. You can, however, if you find a friend who has a SuperDrive. That person can extract the files and copy them to 800 K disks for you.

To get the software from the disk on to your hard drive, follow these simple steps:

1. Insert the disk into your 1.4 Mb floppy drive.

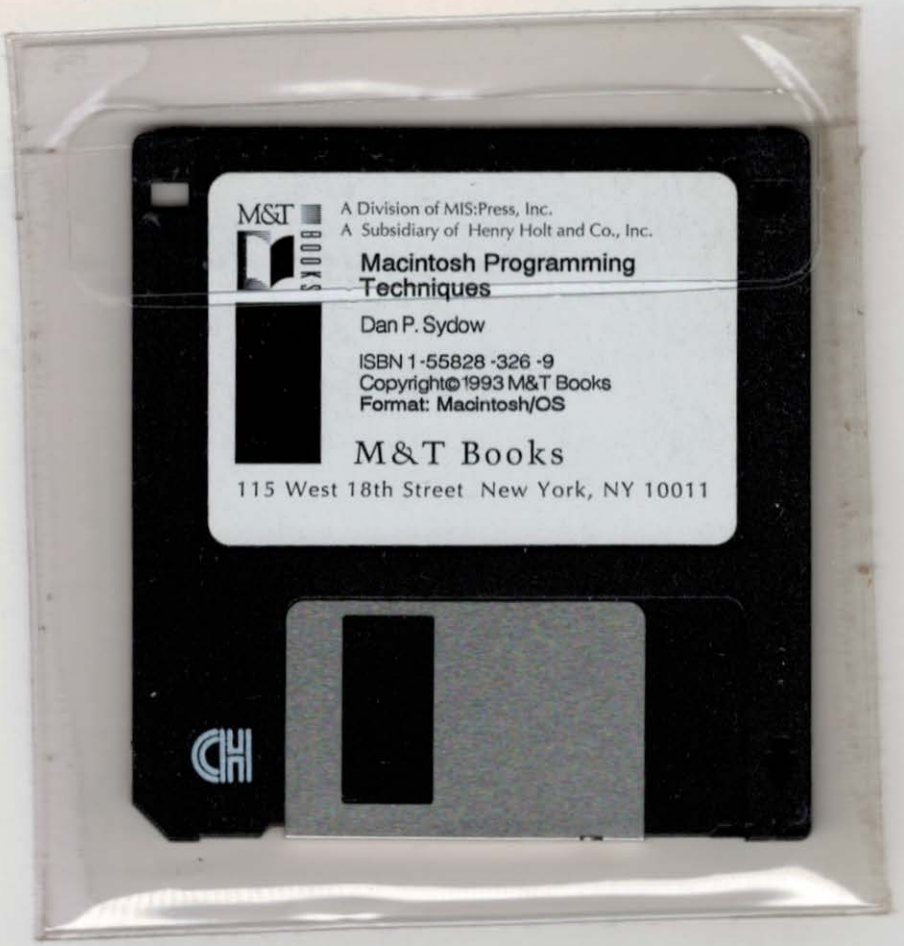
2. Copy the single file, named *MacProgTech.sea,* to your hard drive.

3. Double-click on the *MacProgTech.sea* icon on your hard drive.

4. A dialog box opens. Click the Extract button.

The In Action! Mac Techniques software runs on any Macintosh running System 6.0.4 or greater—including any version of System 7. The program runs on black and white or color systems, and runs on a Macintosh with any size monitor. You need no additional software to run the program.

For more detailed instructions on how to decompress the software, and how to use it, read the section titled "Introduction" beginning on page 3.


M&T BOOKS
A Division of MIS:Press, Inc.
A Subsidiary of Henry Holt and Co., Inc.
Macintosh Programming Techniques
Dan P. Sydow
ISBN 1-55828-326-9
Copyright©1993 M&T Books
Format: Macintosh/OS
M&T Books
115 West 18th Street New York, NY 10011

Master the fundamentals

of Macintosh programming with this hands-on guide and tutorial. It provides a solid foundation for developing powerful applications. No matter what language you use you'll benefit from the dozens of techniques presented. This book contains in-depth discussions of key topics every programmer should know, including memory management, QuickDraw graphics, and event-driven programming.

Macintosh Programming Techniques also includes an interactive software tutorial. Work through the examples to develop an exciting application loaded with the features expected of a Macintosh program — graphics, text, color, and animation. If you're new to Macintosh programming or want to boost your Macintosh programming skills, this is *the* reference for you!

- Learn the tricks and techniques of Macintosh programming
- Gain hands-on experience with an interactive software tutorial provided on the disk
- Discover guidelines for creating consistent and friendly user interfaces
- Use QuickDraw and PICT resources to animate your program
- Understand the Macintosh's memory management system and avoid memory pitfalls
- Examine the resources essential to creating standalone Macintosh programs
- DOS, Windows, and mainframe programmers — make a smooth transition to Macintosh programming

Why this book is for you—page 1.

Dan Parks Sydow has owned and programmed Macintosh computers since the first 128K Macintosh was introduced a decade ago. He is a software engineer at St. Luke's Medical Center in Milwaukee, Wisconsin, where he works on software for the hospital's intensive care units and nuclear imaging department. His educational Macintosh software is distributed by companies such as Intellimation.

The enclosed disk contains a compiled executable version of the software tutorial plus C and C++ source code examples.

US$ 34.95
CAN$ 43.95

ISBN 1-55828-326-9

M&T BOOKS
M & T Books
115 West 18th Street
New York, NY 10011

LEVEL	Basic/Intermediate
TOPIC	Programming
SOFTWARE	Symantec C, C++
HARDWARE	Macintosh

90000>
9 781558 283268