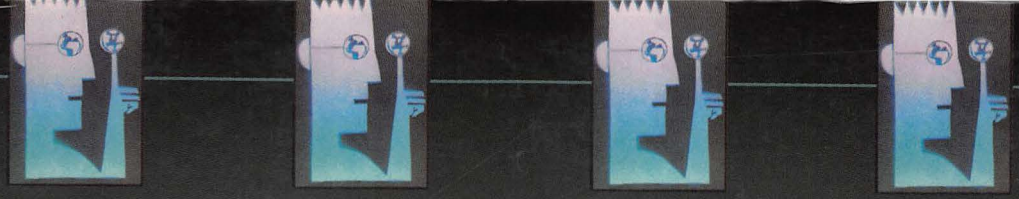


Macintosh Game Programming Techniques

- A complete guide to creating games for the Macintosh
- Covers sound, animation, and game intelligence
- Includes complete source code and guidelines for building Desert Trek
- CD-ROM also contains a limited version of Symantec C++ for the Power Macintosh

CARY TORKELSON

Macintosh Game Programming Techniques

Cary Torkelson

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

Library of Congress Cataloging-in-Publication Data

Torkelson, Cary.
Macintosh games programming / Cary Torkelson.
p. cm.
ISBN 1-55851-461-9
1. Computer games--Programming. 2. Macintosh (Computer)--Programming. I. Title.
QA76.76.C672T67 1996
794.8'15265--dc20 96-19483
CIP

10 9 8 7 6 5 4 3 2 1

Associate Publisher: Paul Farrell
Managing Editor: Cary Sullivan
Editor: Michael Sprague
Copy Editor: Suzanne Ingrao
Copy Edit Manager: Shari Chappell
Production Editor: Joseph McPartland

CONTENTS

ACKNOWLEDGMENTS

This book would not have been possible without the influence and guidance provided by several fine people.

First of all, I would like extend a special thanks to Stephen Dacek and Arthur Goikhman of Soft & GUI Inc. Their vast knowledge and experience has profoundly affected my professional career as well as my programming style.

I would also like to thank Bob Nordling, president of the National Home and School Macintosh Users Group. Over the years, Bob has been a great help to all educational and entertainment shareware authors. His ideas and programs to promote the exposure of shareware authors encouraged me to become more heavily involved in the Macintosh shareware community. Without that involvement, I probably wouldn't be writing this book.

Finally, I would like to thank Glenn Seemann for his help in beta-testing Desert Trek and for his useful comments concerning the content of this book.

INTRODUCTION

Who Will Want to Read this Book?

Macintosh Game Programming Techniques teaches you, in an easygoing, enjoyable manner, how to write programs for the Macintosh computer using the C programming language. Many of the topics focus on techniques required to write quality games, but most of the material applies equally as well to writing any type of Macintosh program. This book assumes that you are already familiar with the C programming language, but don't worry, you need not be a "guru" to glean useful knowledge from this book. The main ideas here are to enjoy learning how to program for the Macintosh, and to help you design and market your games in addition to learning the technical skills required to produce a successful game.

The topics discussed in this book will appeal to a variety of people. Programmers without any Macintosh programming experience will find brief introductions to Mac programming techniques that will allow them to grasp the more advanced principals without much difficulty. Hobbyists, or part-time programmers with "informal" Macintosh programming experience will learn numerous tips and tricks to writing and distributing that successful shareware game. Experienced or professional programmers will find out how I approach and solve specific Macintosh programming issues. Some of my techniques are quite unique, and you'll either learn something new or have a good time laughing at my style.

Why Write a Game?

There are many reasons to write a game for the Macintosh computer, many of which aren't that obvious. First, and most obvious, it might how you make your living. However, you don't have to write games to make a living programming for the Macintosh. For that matter, like me, you might make a living programming for a completely different computer platform. In those cases, you might want to write a game to improve or enhance your programming skills. There's no better way to enhance your programming skills than by writing a game. You'll be motivated, and you will have fun.

Perhaps you want to build a reputation for yourself in the Macintosh programming community, or try to make a little money. You could do so by writing shareware games. There are very few authors who currently make a living writing shareware games, and their style of distribution more closely resembles that of a commercial software company than the typical shareware author. However, there are plenty of examples of shareware authors going on to write commercial games. They built a reputation for themselves writing shareware games, and made the transition to commercial software development using that reputation.

Finally, maybe you just want to have some fun and spread a little joy into the lives of others. Making the world a better place is not easy for ordinary people like you and me. You might not affect millions of people, or change societal attitudes, but you could make one or two people smile by writing a game and sharing it with others. What more could you want?

The Macintosh Game Market

Almost everybody with a computer plays games. They may not write, create databases, number crunch, surf the net, do homework, or keep the family finances, but, with rare exception, they always play games. The market for computer entertainment is wide open. Though the life of a particular game is usually short lived, new games enter the market every month. The same can't be said about word processors, spreadsheets, programming environments, or database programs. This means that you probably won't become financially secure with one game, but you stand a better chance at getting a game to market than other types of programs.

In addition to the large market of commercial games, shareware games abound. More and more CD-ROM collections featuring shareware games are released each year. Again, it's easy to make your mark, get feedback, or just make industry connections writing a game. Almost anyone will try your game (it's up to you to write one good enough to hold their interest), whereas not everyone is interested in utilities, system extensions, and the like.

Lastly, don't overlook the porting market. Many games are ported from the IBM PC compatible. Let's be honest here. The PC game market is much larger than the Mac market. However, more isn't always better. In other words, the are also more bad PC games than bad Macintosh games, and those porting the games from the PC to Mac aren't always experienced Mac programmers or even users. There's plenty of opportunity for Mac programmers in the games porting market.

What Are You Going to Learn?

C Programming Techniques

Even experienced C programmers can stand to learn a thing or two about general programming techniques. I personally have learned a lot working on Wall Street, where your program had better work absolutely perfectly 24 hours a day, seven days a week, otherwise millions of dollars will be lost (not to mention your job).

Program Design and Marketing

Having great Macintosh programming skills doesn't do you a lot of good if you can't design and complete a game, or if you can't market it to the public. Design can make the difference between writing a great game, or producing an "okay, it works but it's not very elegant" game. Are you going to want your game to be available in different languages? Well, you'd better think about that before you've almost finished coding. Don't worry, I'm not going to harp on methodologies, flow charts, or academic garble. The goal is to make design an integral part of the game development process without causing too much pain.

After you've written the game of the century, what do you do with it? You need to get it into the hands of the game-playing public, and there are many ways to do this, depending on your ultimate goals.

The Tools Needed to Write Mac Games

Just exactly what are you going to need to write that cool game? Is your current Macintosh sufficient? How about a compiler and development system? All the hardware and software tools needed to write a complete game are discussed, as well as where to get them.

Macintosh Toolbox Essentials

What is the legendary Macintosh Toolbox, and is it going to help you write that game? You bet it is, since it's chock full of the routines you need to make your game a true Macintosh program. We'll start out with the basics such as events, memory management, menus, windows, dialog boxes, textedit, file I/O, and QuickDraw. Special attention will be given to 32-bit color QuickDraw, and the topics of offscreen graphics worlds, copybits, and copymask (pixel transfer routines). Special attention will also be given to modeless dialog boxes, a topic on which many tread lightly. There's nothing mysterious about them, and I think you'll want to use them all the time once you know their advantages and easy use. Other topics that will be covered include digitized sound, styled textedit records, saving TeachText files with embedded graphics, and automatically loading a file that was double-clicked in the Finder.

Resources

Easily half of your development time will be spent creating, editing, and manipulating resources. Some of the resources covered in this book include 'BNDL', 'CDEF', 'cicn', 'CNTL', 'dctb', 'DITL', 'DLOG', 'FREF', 'ictb', 'MBAR', 'MENU', 'PAT', 'PICT', 'SIZE', 'snd', 'STR#', 'styl', 'TEXT', 'vers', and 'WIND'. You'll also learn how to create custom resources for storing and retrieving program data. Special attention will be given to creating 'ictb' resources which are used to customize the color and text attributes of dialog elements. Plenty of clear examples will be given, something typically lacking in other documentation.

Where to Get Additional Information

I wish this one book could give you everything you need. However, it would take me forever to write, and you probably wouldn't be able to

carry it home. After reading this book, you'll definitely be able to write great Macintosh games, but you'll probably want to seek additional information on certain topics. I'll get you started by telling you what's available and where to get it.

The Bottom Line

This book presents to you all the necessary information to begin writing games for the Macintosh computer. The entire source code for Desert Trek, a popular shareware game that I've written, is included. Nearly all coding examples will be drawn from this program so you can see how everything fits together. Additionally, you'll have a wide set of ready-to-use routines to plug in or adapt to your games.

You'll learn, step by step, how to design a successful game, create all the necessary resources, code, and market your product. Most importantly, you'll have fun!

CHAPTER 1

GETTING STARTED

This chapter will tell you everything you'll need to start writing a successful Macintosh game. It will also cover the programming conventions and styles used throughout this book. It pays to spend a little time thinking about the game you're going to write and how you're going to approach the issues involved before you sit down and start coding. Spending a couple of days early on could shave weeks off the development cycle.

Programming Style

C versus C++

C is a very powerful and flexible language that allows you to implement almost any algorithm efficiently. However, being so flexible, C will let you write code that doesn't do exactly what you wanted, blowing away memory or introducing program bugs that are extremely difficult to track down. In other words, a system error caused by a bad pointer or memory operation may not occur near the origin of the problem.

In addition to other things, C++, an object-oriented extension of the C programming language, attempts to reduce the types of errors caused by poor programming techniques. Like Pascal, C++ forces strong adherence to type rules, reducing the chance that an unintended operation takes place.

Another common debugging problem with C code is locating where a variable receives a bad value. C++ tends to reduce the severity of this problem by separating variables and the code that affects them into objects. Those functions, called *methods*, are typically the only code that can affect the value of the variable, simplifying the debugging process. By its nature, C++ also encourages the practice of reusing code. In fact, several class libraries available for your use eliminate the need to write many operations yourself, especially for the user interface.

So, with all the advantages C++ has over C, why not write Macintosh games using C++? Well, you could definitely do so. However, for two reasons, I believe that C is currently the better language for writing a game. First, the advantages of using C++ come at a cost. (Since when do you get something for nothing in this world?) The cost includes longer training time to learn the language and a longer design cycle. Designing class objects is no trivial task, and care must be taken to do things correctly. Using C++ does not guarantee the creation of well-structured, or even reusable, code. Using prebuilt class libraries to reduce the need to write code yourself also comes at a cost. Class libraries tend to be large because they need to cover a wide variety of applications. Being so

generic, they are usually not optimized for your specific needs and tend to execute more slowly. Program speed and size are very important issues for Macintosh games, and C++ doesn't necessarily help here.

Structure of an "Object-Oriented" C Program

Even though we are forgoing the use of C++, there are many things that we can do as C programmers to benefit from C++ concepts. First, using the object metaphor, we can isolate data and the functions that operate on that data into their own separate units. Global variables used across units are not allowed, because by design, global variables can be altered anywhere. In other words, all variables defined for a particular unit are *private*, meaning that no other unit can directly read or write that variable. In addition, most functions that operate on that data are also private, meaning that they're not callable from other units. Realizing that it is sometimes necessary for one unit to access data from another unit, we can declare public functions callable from other units. These public functions will be declared in the unit's header file. Structures and constants can also be defined in the header file to provide other units with a variable's type information.

Header File (.H)

"Public" constants

"Public" structures

"Public" function prototypes

Source File (.C)

"Private" constants

"Private" structures

"Private" function prototypes

"Public" and "private" function definitions

In C, there really is no formal definition of public and private elements. Typically, functions and variables are considered public since they can, by default, be accessed by other units. Yes, you would need to use the **extern** keyword to access variables from another unit, but functions require no such keyword. In order to ensure all elements defined in a source file are private to that source file, declare all functions and variables with the **static** keyword. All elements declared with the **static** keyword are limited in scope to the source file in which they are declared, making them unavailable to other units.

What about public elements? Remember, we aren't allowed to use global variables, so there are no public variables. Public functions require that the function prototype be declared in the unit's header file, which can then be included by other units requiring access to those functions. Access to variables from other units is always accomplished through function calls to that unit. So, if you need to perform a calculation using a variable from another unit, call a function from that other unit that returns the variable's current value. If you need to set a variable located in another unit, call a public function from the other unit, passing the new value. Note that sometimes the variable one unit wishes to read from another unit is actually a structure. Since it is not efficient to pass entire structures around on the stack, pointers to the structure can be used. However, you're on your honor not to modify that structure in the nonowning unit, which would be possible because you'd have a pointer to that structure. Or, if you want to be really strict, you can require that the calling routineprovide a pointer to storage in which a copy of the structure can be returned. This is actually the preferred, more structured method, but it does have the penalty of a memory copy operation, increasing program execution time.

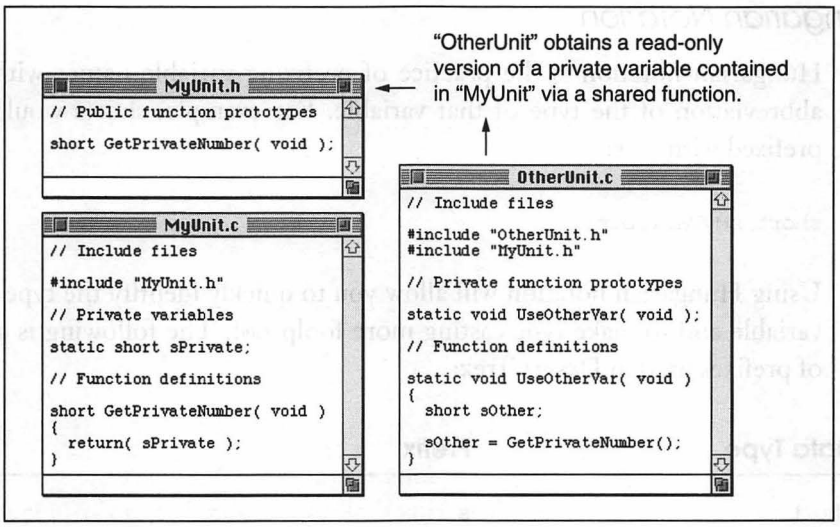


Figure 1.1 Public and private elements and an example of one unit accessing a variable from another unit.

Using the program structure defined here gives C programmers some of the advantages C++ offers without incurring much of the overhead. By isolating data and functions into separate units, code can easily be reused. One caveat, though: Just as in C++, care must be used to design units that can realistically be plugged into other projects with little or no modification.

Additional Programming Conventions

In addition to using the object-oriented programming style described in the previous section, there are several rules that, if followed, help you create more stable and easy-to-maintain code.

Hungarian Notation

Hungarian notation is the practice of prefixing variable names with an abbreviation of the type of that variable. For example, shorts would be prefixed with an **s**:

```
short sMyVariable;
```

Using Hungarian notation will allow you to quickly identify the type of a variable and to make type casting more foolproof. The following is a list of prefixes used in Desert Trek:

Data Type	Prefix
short	s
long	l
Boolean	b
float	fl
Handle	h
Ptr	p
char	ch
char[]	ach, or sz for null-terminated arrays
char *	psz
Str255	str255
DialogPtr	pDialog
WindowPtr	pWindow
Rect	rect
GrafPtr	pGraf
CGrafPtr	pCGraf
GWorldPtr	pGWorld
GDHandle	hGD
Point	pt
SndChannelPtr	pSndChannel
StringHandle	hString
CursHandle	hCurs
SFReply	sfReply
OSErr	osErr
CharsHandle	hChars
SFTypeList	sfTypeList
PicHandle	hPicture
CIconHandle	hCIcon
ControlHandle	hControl
BitMap	bitmap

`TEHandle`	`hTE`
`Size`	`size`
`MenuHandle`	`hMenu`
`StringPtr`	`pString`
`RGBColor`	`rgbColor`
`PatHandle`	`hPat`
`Pattern *`	`pPattern`

Avoid Using the int Type

The maximum value of the int variable type depends on the compiler implementation. As long as your code stays on the Macintosh, things will probably be fine. However, move that portable C code to another platform, and things might be different. If you want a 16-bit integer, use `short`. If you want a 32-bit integer, use `long`. No ambiguity there.

Separate Nonuser Interface Code

To be honest, most of the code written for a game is user-interface related. However, there are parts of the code that deal with the rules of the game and need not be platform-specific. If you plan to port your game to another platform, such as Microsoft Windows, the nonuser interface–specific code can be used with little or no modification. Keeping it separate will help.

What do You Need to Write a Game?

Macintosh Computer

This is kind of obvious, but how much horsepower do you really need? Today, even entry-level Macintoshes are adequate for writing games; in the past, this was not the case. One would have had to spend a large sum of money purchasing a high-end Mac to write even a fairly simple game. Today, any Macintosh with a 68030 processor is more than enough. Of course, if you plan on using high-intensity graphics such as ray tracing tools, your needs may be better met with a PowerMac. Slower computers will certainly not be speed demons when it comes to compiling your

code, but I used to develop games on a Mac SE with a lowly 8 MHz 68000. If you want to write native PowerMac code, you should have a PowerMac. A color monitor is necessary, because your game must support color. You'll need at least 8 MB RAM, but 16 MB is strongly recommend to run several development tools at the same time. A CD-ROM drive is also strongly recommended because high-end development tools and programming documentation frequently come on CD-ROMs. One of the best ways to determine if your Mac has what it takes to write a game is to look at the system requirements of the development system you choose to use. For example, Think C 6.0 only requires a Macintosh running system 6.0.4 or higher and 4 MB of RAM. CodeWarrior Bronze requires a Macintosh with a 68020 or higher and 8 MB RAM running system 7.1 or higher. See Appendix A for more information on these products and where to get them.

Compiler/Development Environment

Typically, compilers such as Metrowerks CodeWarrior or Symantec's C, C++, and Pascal include the development environment. They include an editor, debugger, and project management tools, as well as the compiler itself. Visual design tools are also a standard part of the high-end C++ development environments, but they are typically not appropriate for use on games. Again, see Appendix A for details on where to obtain these products.

Additional Tools

In addition to the development environment, you'll need tools such as ResEdit and MacsBug. ResEdit allows you to graphically edit resource files. MacsBug is a system-level debugger that catches system errors that may not be detected by the development environment's debugger. It is also useful when running your game outside the development environment. You will also need a graphics program to develop the graphics used

in your game. Depending on your needs or ambitions, you could get by with a simple paint program, or you may require a 3D rendering tool. You will also need tools for recording and editing sound. Fortunately, many Macs come with digitized sound input hardware and software. If yours does not, seek out a friend, purchase additional hardware, or resign yourself to using prerecorded libraries (making sure you have rights to distribute a game using those sounds).

Internet or Online Service Access

Though not mandatory, Internet access, or an online service such as America OnLine or CompuServe will be extremely helpful during all stages of development. Browsing game-related newsgroups can give you ideas on what types of games would be popular or successful in the market. Programming-related newsgroups will prove invaluable when you have programming questions. Nothing beats the advice of a live, experienced programmer when you need help.

You'll need beta-testers to ensure a quality game without too many bugs. Again, it's fairly easy to find many beta-testers with a wide variety of Macintosh equipment on the Internet. A game that works perfectly on your Mac may totally bomb on another. Also, many good ideas and suggestions come from those who aren't as enamored with your game as you are. A word of wisdom: take feedback from beta-testers very seriously. Your game will be improved by implementing their suggestions. Remember, it's users like them who will be deciding whether your game is good enough to pay for.

Lastly, when you have a completed game on your hands, you'll need to announce its presence to the world. Whether you'll be distributing your work as shareware or publishing it commercially, nothing beats the advertising and distribution you can get with the Internet. Obviously, if your game is shareware, distribution via the Internet is necessary to get it in the hands of the public. Even if you've written a commercial game, you'll want to distribute a demo version to spark interest.

Game Design

I'm sure you have a great game idea and want to start writing it immediately. However, spending a little time up front figuring out the details will pay off big by reducing overall development time. It's especially important to determine your goals for the game up front. Do you want to commercially distribute the game or release it as shareware? You'll probably have more fun writing a shareware game since it doesn't have to be as polished and perfect as a commercial game. That doesn't mean that there aren't shareware games that look and play much better than some commercial games. However, there's nothing like having to do something to eat and pay rent to take some of the fun out of it.

Defining Your Target Market

What type of game do you want to write, and will it be successful in the market? There's a wide variety of games out there: arcade shoot 'em ups, role-playing adventures, sports games, card games, puzzle games, strategic simulations, and so on. It's important that you choose something you're really interested in because that will improve the chances you'll finish it successfully. However, give some thought to how popular it would be in the market. Take a look at recent popular shareware and commercial games to get an idea of what's doing well. Browse several online newsgroups to see if there's a gap in the market. For example, there are frequent complaints in the Mac community concerning the lack of decent sports games. Arcade games tend to be very popular, but that popularity is typically short-lived. Strategic simulations usually have a smaller following, but those who play them are usually very supportive.

Which Mac Platforms to Support

Decide up front which Macintosh platforms you intend to support. Will your game require 8-bit color, or can it run on black-and-white displays. Will you support smaller screens, such as the 12-inch monitor? Are you

going to have a native PowerMac version? Will you support 680x0-based Macs? Many games today require at least a 68020 processor. Lastly, will your game require system 7, or will it support earlier versions? The answers to these questions determines the size of the market your game will cater to, and the complexity of your code. The more platforms you support, the larger your potential market, but you'll need to spend extra time coding and testing on all supported platforms. Figuring out up front which Macintosh platforms to support will allow you to design your game appropriately. For example, if you plan to support system versions prior to 6.0.7, you'll need to disable any digitized sound routines when running on those systems because those sound routines require system version 6.07 or later.

Make it Macintosh

It's easy to say, "Hey, this is a game and I can completely take over the machine and build whatever interface I like." That may have worked in the DOS world, but you'd be asking for trouble from the Mac community. Your game is going to run on machines that occasionally have to do real work (no, really), and your game will, if not cooperate with other programs, at least have to leave them unaffected.

This means you'll need standard Macintosh elements, such as a menu bar and movable windows. Many games like to completely take over the screen, hiding the menu bar. There's nothing wrong with that, as long as you provide an easy way to show the menu bar, quit the game, and switch to other programs that may be running. Try to use standard Macintosh controls when possible, because users will already be familiar with them.

If you happen to be porting a DOS game, do *not* under any circumstances try to emulate a custom full-screen menu, use 8.3 filenames, or ask users what kind of monitor they have when installing your game. In addition, porting "chunky" 320x200 graphics directly without at least some smoothing generally leaves a bad taste in users' mouths. The bottom line is this: you are writing a Macintosh program; make it a Macintosh program.

Pick a Project You Can Finish

One of the biggest reasons software projects, especially games, fail is that the author attempts to do too much. This rings doubly true if this is your first game or if you plan to release the game as shareware. Now, every one of us wants to write that totally awesome, all-encompassing, market-busting game, but let's get real. If you don't have a multimillion-dollar company providing the resources, you may want to scale back your goals. A game can be popular and fun to play without full-motion video. In fact, full-motion video won't make a bad game good.

For individuals who are writing their first game, another reason for choosing a modest project is coding style. Your first game will not have the best code. That's okay since as with anything else, practice and experience will improve your style. However, if you choose a large first project, you'll notice that code written early in the project tends to be of much poorer quality than code written near the end of the project. Sometimes consistent mediocre code is better than inconsistent code because maintenance and debugging will go much more smoothly if the code is fairly consistent.

Reusing Code from Previous Projects

Writing your first game will take much longer than writing subsequent games because you have no code base to use. After writing your first game, you should have hundreds of lines of code that can be used in subsequent games, either directly or with small modifications. Again, good design on that first game will provide you with a lot of reusable code, reducing the time it takes to write the next game. As a general rule of thumb, make routines as generic as possible so they can be used for other projects. However, do not do so at the expense of performance or the look and feel of the game you're currently working on. After writing several games, you should have a number of routines that allow you to build a skeleton application in very little time.

The entire source code for Desert Trek is included on the CD-ROMwith this book. This means that you have a number of routines that are ready to be plugged into your games with little or no modification.

Hopefully your first game will take less time to write than my first game did. That's one of the goals of this book.

Finish the Design Before You Code

I know, you really want to get some cool stuff up and running as fast as possible. There's nothing wrong with that, but don't rush into programming without a solid design in hand. This doesn't mean you need a twenty-page thesis describing the technical details of the game. Maybe it just means you've written a few notes on a scrap of paper or just sat down to think about what you want the game to become. Doing so will make coding the game progress more quickly and will require less rewriting of code due to design changes. Lastly, having a complete design will help you determine the order in which to code things, making testing easier.

The Design of Desert Trek

To practice what I preach, so to say, I'll take you through the design of Desert Trek to give you a look at my experience that designing a game is not painful and provides plenty of benefits.

A Game Idea is Born

Okay, so I'm sitting there thinking to myself, "I want to write a game." What kind of game? Well, I'd like to write a small game, in terms of size and resources, that's fun to play as a little diversion now and then. We're not talking a major game that requires days to learn, hours to play, and a big chunk of your hard drive. I'm ruling out arcade games because I don't feel like writing one right now. Actually, my specialty is strategy, puzzle, or logic games, and I feel comfortable writing them. I'm going to make my game shareware, so I don't need to worry about marketing or deadlines. Now I need an idea that fits these criteria.

Back in the early days of personal computing, numerous simple text-based games proliferated. Many of them were written in Basic, and you had to type them in yourself if you wanted to play. Actually, it was a great

way to learn computer programming, because typing in all that code was bound to rub off on you after a while. One such game I vaguely but fondly remember was a game called Camel. The object of Camel was to ride a camel across the Gobi Desert. It was simple but fun, and I thought an expanded Macintosh version with a healthy dose of graphics would do well as shareware. Desert Trek is born.

Game Rules

Before playing a game, you need to know the rules. The same applies to writing a game.

The goal of Desert Trek is to travel 1000 kilometers across the Gobi Desert. Pretty simple, huh? My challenge as a game designer is to throw in enough challenges to prevent the game from becoming boring quickly while maintaining its simplicity. First, we need some hazards to prevent the game from being too easy to win. The original Camel had several: hungry cannibals chasing you, wild Berbers hidden in the sands wishing to capture you, sandstorms, the constant threat of death by thirst, and the possibility of running your poor camel to death. I, of course, kept these hazards and added several more: hunger, camel health, dangerous paths in the sands, and unfriendly caravans wishing to steal your precious supplies. In order to help the player overcome these hazards, the original Camel had oases to rest your camel and replenish supplies. In my version, I added friendly caravans, abandoned campsites where you could find money, and trading posts that sold useful supplies such as a compass to help you find your way in a sandstorm and binoculars to better see what's ahead.

For the complete playing rules of Desert Trek, you can read the Help section found in the program. However, from a game designer's point of view, simply stating the game's playing rules is far from enough information to write the game. There's much to be decided that takes place behind the scenes that the players generally don't need to know. Before you continue reading the rest of this section, you may want to familiarize yourself with the Desert Trek game rules. Familiarity with the game rules will help you understand some of the design decisions I made.

Internal Design Issues

Let's examine more closely the danger of thirst. In the original game, thirst increased by a set amount per turn. After a certain point, the player would perish unless he or she drank water. For my implementation of Desert Trek, I wanted something a little more complex to make the game less predictable or to at least require the player to pay more attention. To accomplish this goal, I wanted thirst to increase by different amounts. I came up with two methods in which to implement this. First, instead of the game consisting of a number of turns that were essentially the same, I added the concept of days. In other words, each *day* was broken up into four turns, one each for morning, midday, evening, and night. Now I could have thirst increase twice as much in the middle of the day, when the temperature was supposedly hottest. Though this may seem to be a trivial difference compared to the original game, this and several other time-dependent factors means that players need to plan ahead to win.

Skill Levels

The second method by which I caused thirst to increase at different rates was to add skill levels. In actuality, *skill level* affects all aspects of the game, from how much money you can find at an abandoned campsite to how many kilometers you and the cannibals can travel per turn. Skill levels bring up a very important issue to the game designer: how easy or hard do I make the game to win? If the game is too easy to win, players will quickly tire of winning and stop playing. If it's too hard, players will quickly tire of losing and give up. What may be easy or hard to you will not be the same for the next person. Much thought needs to be put into determining the difficulty of the game. Adding skill levels gives you, the programmer, some flexibility to appeal to a larger variety of players. It adds some work to the game, but in some ways it makes your job easier. Think about this: you create your game, but it's too hard. You need to "tweak" some game parameters to make it easier. This requires coding changes and retesting. Now the game is too easy. This requires more tweaking, recompiling, and testing.

After a few iterations, this method starts to become cumbersome. One way to overcome this process is to store game parameters in the resource file. You only need to change a resource to affect the difficulty of the game. Now that your game is using data from a resource to drive its difficulty, there's only a small step to providing skill levels. Make a copy of the resource, change a few parameters, and viola, you have a new skill level. Providing skill levels allows game players to choose the level of difficulty and lengthens the time players remain interested in the game. Once they win, they will try again at a higher skill level. Design skill levels so that new players will be able to win at the easiest skill level after a couple of games and so that experienced players aren't always going to win at the hardest. The more skill levels you have, the better, because players of differing abilities will be able to find the skill level that challenges them without making it impossible for them to win. This is a little more work for you, but your game will appeal to a wider variety of players.

To give you an example of the wide variety of playing abilities game players have, let's look at one of my most popular games, Galactic Empire. Like Desert Trek, Galactic Empire has ten skill levels. I designed the skill levels so that I could almost always win at skill level 1, win frequently at skill level 5, and never win at skill level 10. Did I make the game too hard? Well, I have received correspondence from several people stating that I had made the game totally impossible to win. In fact, they were upset that I'd release a game that was so obviously impossible to win even at level 1. On the other hand, I have also received correspondence from several users who have won at skill level 10, a skill level that I admit to not being able to win (and I wrote the game!). Based on the variety of responses, I feel good about the balance. If everyone won at level 10, the game would have been considered much too easy. If nobody could win at level 1, the game would be considered much too hard.

I admit that some types of games lend themselves better to supporting a variety of skill levels than others. However, skill levels don't have to be of the "select from 1 to 10" variety to be effective. For example, a role-

playing game may make the first few quests easy and progressively increase the difficulty of the quests as the player progresses.

Other Features

After defining the game rules and coming up with general ideas on how you're going to implement them, you need to think about what features your game will support. One of the most common features for games, other than those of the arcade variety, is the ability to save and load games in progress. Another popular feature is a high scores list. Lastly, all games need to provide help to the players. Typically, this is either provided as a separate **readme** text file or integrated into the game. There are benefits to both methods. A separate text file gives the users the ability to import the text into their favorite word processor and print the instructions. However, integrated help allows the user to easily get help while playing the game and eliminates the need to distribute separate files, which could become separated somewhere in distribution.

Desert Trek supports these features and provides a slight twist on each. First, in addition to saving and loading games, the player can save the journal as a text file. Though this isn't something to write home about, it allows game players to print a record of their game by loading that text file into a word processor or Teachtext (now called SimpleText). The top 10 high scores are kept for each skill level. In addition to maintaining high scores, Desert Trek allows users to export the high scores into a file, which can then be imported into a friend's copy or a later version of Desert Trek. When importing scores, the user can overwrite the current scores or merge the two lists, keeping the highest scores from both. I believe that some method of transferring high scores is especially important for shareware games. When a user registers, I send out the latest version of the game. Without this feature, the user would lose their high scores—not a good reward for registering. Finally, the help I provide with Desert Trek is included within the game, but an option to save the instructions as a text

file is provided. This eliminates the need for distributing a separate file while giving the user the ability to print the game instructions.

Screen Layout

It is very important to design the layout of the game screen or screens before you get too far along in the project. Probably the most important aspect of game design, the screen layouts define how the user will interact with your artificial world. The screens should be easy to read, not too cluttered, look pleasing, and follow standard Macintosh user-interface guidelines. It doesn't matter how great your game is; if the screens don't look cool, you can forget about attracting many players. However, don't let the opposite occur either; there are many games with great colorful graphics, but little substance.

For Desert Trek's game screen, of which there is only one, I wanted to display certain information. First and foremost, the game player needs to know two very important things: status and supplies. This includes hunger, thirst, camel fatigue and health, distance traveled, cannibals distance, and how much food, water, and elixirs you have. In addition to these essentials, the player needs to know what time of day it is, because that greatly affects certain aspects of the game. I accomplish this by providing a simple first-person view of the desert. Based on the position of the sun, the player can plainly see what time of day it is. In addition to being able to show the player the time of day, the first-person view shows the player what objects can be seen in the distance. When there's an oasis in the distance, a picture of the oasis can be seen in the view portion of the game window.

Also displayed in the game window is a textual journal of the player's trek across the desert. This journal logs everything the player does and sees and allows him or her to review the history of the game by scrolling a standard text edit box. Lastly, I wanted 3D picture buttons on the main game screen to allow the user to enter game commands without going to the menus. Everything needed to play a complete game of Desert Trek can be found on the game screen.

Just knowing everything that's going to be displayed on the game screen isn't enough. You also need to decide where to place things so that the screen doesn't look cluttered and so that it is possible to find everything easily. You'd be surprised how little things like button placement can greatly improve the playability of a game. For Desert Trek, I grouped related gauges and buttons together. For example, the gauges for hunger and food and the button for eat are placed next to each other. Also, pay attention to the little details. The distance indicators could easily have been textual. However, by adding the graphic thermometer, which displays distance traveled for both the player and the cannibals, the player can quickly assess the status of the game.

Since screen layout is vital to the design of the game, I always use a graphics program to draw what I envision the game screen will look like before I even think of coding. A good portion of your game code will deal with the display of game information, so knowing what you're going to do always helps. Drawing the screen before coding also allows you to position everything perfectly without coding it by trial and error.

To be sure, Desert Trek lends itself to a nice, organized, compact game screen; that's how the game was designed, and by completing the design up front, I was able to elegantly integrate game play and screen layout.

Beyond the main game screen, there are several dialog boxes, a high scores screen, and two information windows to be found in Desert Trek. Again, attention to detail with the auxiliary screens will pay off with a polished, professional looking game. Judicious use of color for the dialog boxes adds a lot to the game, but don't overdo the colors; the results will be gaudy and will detract from the game. Using a light gray background, dark colors for text, light background colors for text entry fields, and 3D push buttons make for a nice effect.

CHAPTER 2

MAC TOOLBOX BASICS: MEMORY AND EVENTS

The Macintosh provides a large set of application programming interface calls (commonly referred to as APIs) to aide you in writing Macintosh programs. These APIs are collectively known as toolbox calls, and when used correctly, allow all Macintosh programs to have the same style. Every Macintosh contains the toolbox physically located in ROM chips built into the machine. It is this legendary toolbox that has, until recently, prevented clone-makers from creating Macintosh clones. Without the toolbox, the Macintosh is useless. Recently, though, Apple has licensed the toolbox to clone manufacturers to increase Macintosh market share against the Intel-based IBM PC and compatible machines.

Not all APIs described in this book reside in the toolbox. Some calls reside in the system software. However, for most purposes, it is unnecessary to distinguish between the two. Related toolbox calls are grouped into *managers*. For example, all menu-related routines make up the menu manager. The number of managers and APIs are so extensive it would be impossible to cover them all in one book. The following is a list of managers discussed in this book:

1. **Operating System Utilities.** The operating system utilities include routines to help determine the Macintosh environment (system version, whether or not certain APIs that are available on this particular Macintosh, etc.). Not all operating system utilities are in ROM. Some are part of the Macintosh operating system itself. Many of these routines are described later in this chapter.
2. **Memory Manager.** The memory manager contains routines to allocate and manipulate memory and strings. You will use these routines to dynamically allocate storage for your game. These routines are discussed later in this chapter.
3. **The Events Manager.** The events manager contains routines to handle user and system events, such as mouse clicks, keystrokes, and update events (when a window's contents need to be redrawn). Because Macintosh programs are event driven, it is very important to understand these routines. They are discussed in detail later in this chapter.
4. **The Resource Manager.** The resource manager contains routines to load, save, and change *resources*. Resources are objects that your program uses, such as menus, dialogs, and pictures. Chapter 3 discusses resources in detail.
5. **Window Manager.** The window manager contains routines related to, you guessed it, windows. Window manager routines are used to show, move, size, and update windows (the actual drawing routines themselves are part of Quickdraw). Chapter 4 discusses windows in detail.
6. **Menu Manager.** The menu manager contains all routines related to the Macintosh menu bar. Routines to insert menus, change their contents, and determine what the user has selected can be found in the menu manager. Chapter 5 discusses menus.

7. **Dialog Manager.** The dialog manager contains routines to handle dialogs and alerts, such as loading dialogs, handling dialog events, and manipulating dialog controls. Chapter 6 discusses dialogs in detail.
8. **Control Manager.** The control manager contains routines related to controls, such as buttons and scrollbars. Routines to load controls, read and set their values, and draw controls can be found in the control manager. Along with dialogs, Chapter 6 discusses controls because controls can be found in every dialog (even if it's just a text control).
9. **Quickdraw.** Quickdraw is one of the most important toolbox managers since it controls everything displayed on the screen. In fact, many of the other managers call QuickDraw routines to perform their tasks. For example, the window manager calls QuickDraw to draw a window's title bar and frame. From the point of view of a game author, QuickDraw is absolutely the most important manager, because graphics tend to play heavily in game design. Chapter 7 provides the details.
10. **TextEdit.** TextEdit provides a set of APIs that allow you to display, format, and edit text. This includes routines to draw text, scroll text, and define text styles such as bold and italics. TextEdit is covered in Chapter 8.
11. **File Manager.** The file manager contains routines to create, read, and write files, as well as routines to present users with the standard file **Open** and **Save** dialog boxes. Chapter 9 describes these routines.
12. **Sound Manager.** The sound manager contains routines to play and manipulate sound on the Macintosh. This includes APIs to create sound channels, play short sound effects, and play background music from sound files. Chapter 10 discusses the sound manager and its routines.

There are a number of managers that are not discussed in this book, such as the appletalk manager, the print manager, and the device manager. For the most part, you won't need these managers to write a game (though if you want to support networking, you'll definitely need to look at the appletalk manager). However, if you do need information about these

managers, see Appendix A for additional information. There just isn't enough room in one book to cover everything, so I'm trying to concentrate on those routines that are necessary for most game functions.

Pascal Considerations

The original Macintosh was designed at a time when Pascal was more popular than C. For this reason, the Macintosh toolbox calls use Pascal calling conventions and string definitions. Since you will be writing C programs, there are a couple of things you'll need to be aware of to deal correctly with the toolbox.

Pascal Calling Conventions

All Macintosh toolbox calls use the Pascal calling convention as opposed to the C calling convention. For the most part, you will not need to worry about the differences because all toolbox calls described in this book are shown in their C incarnation. However, you will eventually look up a Macintosh toolbox call and find it shown in its Pascal incarnation. By knowing the following three rules, and the integer data type conversion table, you will be able to supply your C parameters correctly:

1. For parameters of size 4 bytes and less, use the variable itself.
2. For parameters greater than 4 bytes, use a pointer to that variable.
3. For parameters defined as type **var**, use a pointer to that variable. In Pascal, **var** parameters are parameters passed by reference. Parameters passed without the **var** keyword are passed by value. Remember, parameters passed by value cannot be changed in the called routine, whereas parameters passed by reference can be changed.

Pascal Data Type	C Data Type
`integer`	`short`
`longint`	`long`

All Macintosh data types can be used without conversion since your C compiler will define them correctly for you. How about a few examples to make things clear? I'm not going to describe in any detail the APIs used in these examples because they are covered in other parts of this book. The important thing here is to learn how to convert a Pascal toolbox definition into its C counterpart. Remember that if you get really stuck, you can always open the appropriate header file supplied by your C compiler to see how the toolbox call is prototyped.

In this first example, one parameter is a Macintosh toolbox type, which does not need to be converted. The other parameter, an integer in Pascal, needs to be converted to a short to be used correctly in C:

```
Pascal Definition: function SetVol(  vName      :StringPtr;
                                     vRefNum    :Integer): OSErr;

C Definition:      OSErr SetVol(     StringPtr  pStringName,
                                     short      sRefNum );
```

In the second example, even though the Macintosh toolbox type of Rect does not need to be converted, it is greater than 4 bytes in length (the `Rect` type has a size of 8 bytes). A pointer to the variable must be used when making the C call:

```
Pascal Definition: function PtInRect(  thePoint   :Point;
                                       theRect    :Rect): Boolean;

C Definition:      Boolean PtInRect(   Point      pt,
                                       Rect       *pRect );
```

For the last example, the **var** keyword is used in the Pascal definition. Even though the `Point` data type is 4 bytes in length, the C call must use a pointer to the variable so that its value can be changed in the toolbox call:

```
Pascal Definition: procedure LocalToGlobal( var thePoint:Point);

C Definition:      void      LocalToGlobal( Point *pt );
```

There's one final note about Pascal calling conventions. Occasionally, you will need to supply a toolbox call with a pointer to one your own func-

tions. This, by the way, is known as a *callback* function since the toolbox call, which your code called in the first place, will call back one of your own functions. You should certainly use the conversion rules listed above to make sure your parameters are defined properly, but there's also one additional piece of information you need to know. Make sure to define your function using the **pascal** keyword. The **pascal** keyword in C defines a function using the Pascal calling conventions. In fact, all the toolbox function prototypes defined by your C compiler are defined using the **pascal** keyword. Basically, C and Pascal put function parameters on the stack in different ways. The toolbox routines expect the parameters to use the Pascal method, not the C method. Keep this in mind when prototyping and defining any callback functions because if you leave out the **pascal** keyword, you will most likely cause a system error (or, at the very least, unexpected program behavior). Here's an example (this particular function will be discussed in Chapter 8 on Incorporating Text):

```
pascal void ScrollText(  ControlHandle     hControl,
                         short             sPart );
```

Pascal Strings

The format of a Pascal string differs significantly from that of a C string. A C string is simply a number of characters terminated with a null character (zero, or '\0'). There is no length restriction on a C string; it can be of any size. A Pascal string, on the other hand, has a maximum length of 255 characters. Why? Well, the first byte of a Pascal string isn't the first character of the string itself, but a length indicator. This length byte determines the number of characters contained within the string, so there is no need for a null-terminating byte. Unfortunately for C programmers, most Macintosh toolbox calls that take strings as arguments expect them to use the Pascal format. This can get very annoying when you plan to use standard C string manipulation functions, such as **strcpy()**, **strcat()**, and **strlen()**, and Macintosh toolbox calls on the same string. The C functions expect a null-terminated string whereas the toolbox calls expect a Pascal format string with a first byte length indicator. You will need to convert strings between the C and Pascal formats in order to use both

types of calls. Most Compilers provide routines to accomplish this task, and they will be described later in this chapter's in the section on strings.

Architecture of a Mac Program

So, what does the basic structure of a Macintosh program look like? Here's the **main()** function from Desert Trek:

```
main()
{
  // Initialize the program and if no errors, continue.
  if ( InitDesertTrek() )
  {
    // While we're not quitting, look for and process events.
    while ( !bQuitting )
      CheckEvent();

    // Clean up the program's resources.
    CleanupDesertTrek();
  }

  return( 0 );
}
```

Pretty simple, huh? You're going to have that killer game out the door in no time, right? Well, maybe, maybe not. Basically, all Macintosh programs look for events and respond to them. In other words, Macintosh programs are event driven. For the most part, the user generates the events, such as mouse clicks, menu selections, and keystrokes. In addition to the user, though, the Macintosh operating system itself might generate events for your program, such as window updates (when part of your program window needs to be redrawn).

The other critical aspect of a Macintosh program is that it displays information to the user. This is especially true for a game program. How many computer games have you played that display no information (a sound-only game)? Anyone familiar with the Macintosh knows that information gets displayed in windows. Every program has its own separate window or windows, which can be moved around on the screen to suite

the user's preference. Some games, especially those of the arcade variety, take over the entire screen. However, unless they're writing directly to the video hardware (something that I strongly object to because it's so unMac-like and will cause you nothing but trouble), they are still displaying game information in a window the size of the whole screen. Remember that you should never take control of the entire Macintosh for your game unless you believe that taking over the whole screen is necessary to enhance the atmosphere of your game.

Lastly, you need to be aware that your program isn't the only thing running on the user's Macintosh. Maybe the user is taking a break from work to play your game, and there's something going on in the background. The Macintosh provides a "cooperative" multitasking environment, which means that you need to cooperate with the Macintosh to make sure other programs get processor time when your program is running.

In summary, a Macintosh program does the following:

- Performs initialization.
- While not quitting, checks for events.
- If there's no event, give up some time to the operating system for other tasks, and perform any background processing. (For example, if your program is a real-time game, you may need to move the enemy ship a little bit closer to the player's ship whether the player does anything or not.)
- If there is an event, process it (typically resulting in the display of new information to the user, be it moving a space ship or showing how many widgets are left).
- When the program terminates, perform any cleanup work.

Initializing the Toolbox Managers

Many toolbox managers need to be initialized before you can use their routines. It is a good idea to do this initialization as the very first thing in your game. The following function performs all the toolbox manager initializations for Desert Trek:

```
static void InitToolbox()
{
  InitGraf( &qd.thePort );
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs( 0L );
}
```

Though most of the manager initialization routines take no parameters, notice that `InitGraf()`, the call used to initialize quickdraw, takes a Macintosh quickdraw global variable, `qd.thePort`, as a parameter. The global variable is available to every Macintosh program running on the system, and you do not define it yourself. Chances are, you'll probably never use it in your game anyway. However, you must specify the global variable because quickdraw will most certainly use it.

Memory

Before looking at specific toolbox managers and the routines they provide, we need to take a moment to look at how the Macintosh handles memory. Many of the toolbox calls take or return pointers, handles, and strings. Understanding how these constructs work will make your life as a programmer much easier.

The Stack and Heap

When your program runs, the Finder allocates a fixed memory block in which your program runs. Any memory your program needs for code, variables, and dynamically allocated storage comes from this block allocated to your program. When you build an application, you specify the default and minimum amounts of memory to be allocated to your program. The user can change this setting using the **Get Info** command from the Finder. This memory setting is very important because many system errors are the result of an out of memory condition.

The memory block allocated to your program is divided into two variably sized portions and one fixed portion. These are the stack, heap, and global variables. It is the global variables that remain fixed during program execution. They get stuffed at one end of the memory block. The stack and heap grow as needed during program execution. The stack starts at one end of the memory block, the heap at the other end. They both grow toward the middle as they expand (see Figure 2.1).

The stack stores local variables defined in functions, as well as the parameters passed to those functions. Every time you call a function, the stack grows. When that function returns, the stack shrinks. One of the very nice things about the Macintosh program stack is that there is no fixed maximum size for the stack beyond that of the size of the block of memory allocated to your program. Many other architectures, such as DOS/Windows and OS/2 require that you specify a maximum stack size when you compile your program.

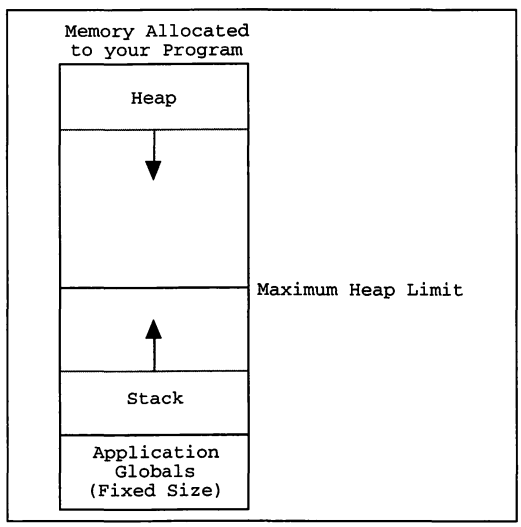


Figure 2.1 The heap and stack memory.

The heap contains memory allocated during program execution. This includes resources loaded and used by your program such as windows, dialog boxes, menus, and even program code itself. Whenever your program

allocates storage at run time, the memory also comes from the heap. Because your program will likely allocate and free blocks of memory many times while it is running, the heap can become fragmented. At some point, this fragmentation could prevent you from allocating a block of memory even if there is enough total memory available. The problem might be that there isn't enough contiguous memory available (see Figure 2.2). When this happens, the Macintosh memory manager can perform heap compaction to collect all the smaller chunks of free memory and create a larger contiguous block. However, in order to do this, the memory your program allocates must be relocatable. Fixed memory allocation prevents the Macintosh memory manager from performing heap compaction on that memory. The following sections on pointers and handles will cover the pros and cons of fixed versus relocatable memory allocation.

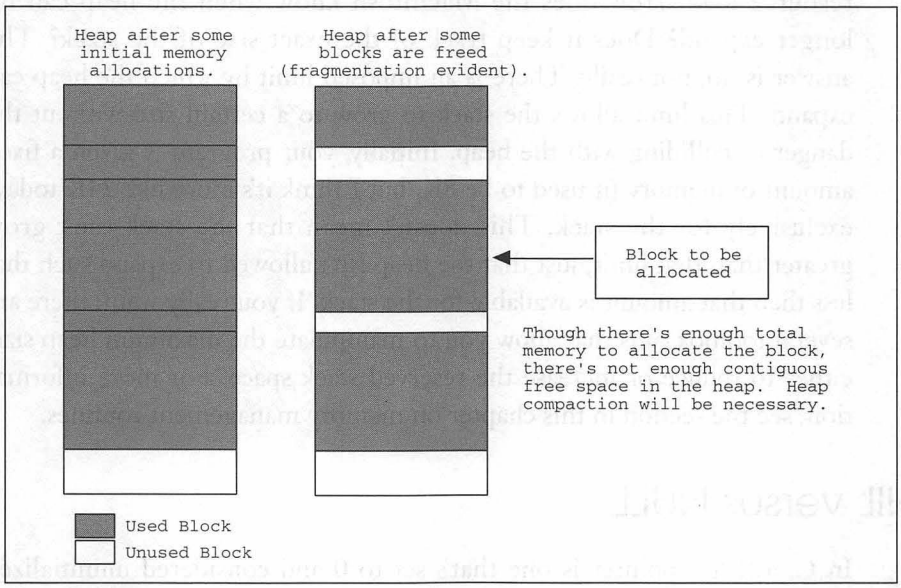


Figure 2.2 Heap fragmentation.

A very interesting and very, very bad thing can happen. Since the stack and heap both grow toward the center of free memory, they can eventually collide. Actually, it is the stack that will always collide with the heap

because the heap won't expand if there isn't enough memory to load a resource or allocate memory. I can guarantee that a heap/stack collision will produce most undesirable results. Make sure to fully test your game to gauge memory usage so that the user never has to find out what terrible things happen when a program runs out of memory. You also need to check to make sure memory you allocate actually gets allocated, and that resource you try to load gets loaded. A simple check on a pointer or handle can prevent many system errors. Sure, a memory allocation error or resource load error makes it difficult to continue program execution, but if you catch it, you can at least exit gracefully.

Here's a little more detail on heap/stack collisions. Remember, I said that the heap won't actually collide with the stack because the heap can't expand if there isn't enough memory available for a memory allocation of resource load. How does the Macintosh know when the heap can no longer expand? Does it keep track of the exact size of the stack? The answer is no, not really. There is an imposed limit by which the heap can expand. This limit allows the stack to grow to a certain size without the danger of colliding with the heap. Initially, your program is given a fixed amount of memory (it used to be 8K, but I think it's more like 24K today) exclusively for the stack. This doesn't mean that the stack can't grow greater than that limit, just that the heap isn't allowed to expand such that less than that amount is available for the stack. If you really want, there are several toolbox calls that allow you to manipulate the maximum heap size, either to reduce or increase the reserved stack space. For more information, see the section in this chapter on memory management routines.

NIL versus NULL

In C, a `NULL` pointer is one that's set to 0 and considered uninitialized (and not pointing to anything). The equivalent in Pascal is `nil`, and, you guessed it, most toolbox calls return `nil` or take `nil` as a parameter when pointers and handles are concerned. To be honest, `nil` and `NULL` are exactly the same thing, so you usually don't need to know the difference. The following `#define` has been defined in Think C/Symantec C++ so that you can use `nil` in your C programs without worry:

```
#define nil NULL
```


Pointers

Macintosh pointers work pretty much like C pointers. Memory allocated using any of the toolbox pointer routines is fixed, meaning that the memory manager cannot move the memory during heap compaction. For this reason, I strongly recommend against allocating pointers in your game. However, several toolbox routines take pointers as arguments, so you do need to know a little about them. The following routines allow you to allocate, manipulate, and free pointers to memory:

```
typedef long Size;

// Allocate a fixed block of memory.
Ptr NewPtr( Size sizeByteCount );

// Free a fixed block of memory.
void DisposPtr( Ptr ptr );

// Get the size of an allocated fixed block of memory.
Size GetPtrSize( Ptr ptr );

// Set the size of an allocated fixed block of memory, which
// can be greater or less than the current size.
void SetPtrSize(  Ptr   ptr,
                  Size  sizeNew );

// Copy memory from one fixed block to another.
void BlockMove(   Ptr   ptrSource,
                  Ptr   ptrDestination,
                  Size  sizeByteCount );
```

Determining Memory Errors

The following toolbox definitions and function, `MemError()`, allow you to check the results of the previous call to a memory allocation routine. Basically, a nonzero result code is bad.

```
#define NoErr           0       // No error.
#define MemFullErr      -108    // Not enough room in heap.
#define NilHandleErr    -109    // Bad operation on empty handle.
#define MemWZErr        -111    // Bad operation on free block.
#define MemPurErr       -112    // Bad operation on locked block.
#define MemLockedErr    -117    // Tried to move a locked block.

typedef short OSErr;

OSErr MemError( void );
```

Handles

Handles are used to allocate relocatable blocks of memory. They are very important to Macintosh applications and should always be used instead of pointers when allocating memory for your game. Basically, a handle is a pointer to a pointer. The pointer that it points to is called a *master pointer.* The master pointer points to the actual block of memory, and the handle your program defines points to the master pointer (see Figure 2.3). Why do all this? Well, a relocatable block of memory might just move when the Macintosh does heap compaction. If the block of memory moved, the pointer to that block of memory must change its value in order to be pointing to the correct location (you don't want it pointing to where the block of memory *used* to be). Because the Macintosh doesn't want to start changing your variables on you, it changes the master pointer, which is exclusively under the Macintosh's control. Your handle remains unchanged because it points to the master pointer, which didn't move. Just the memory block it's pointing to moved.

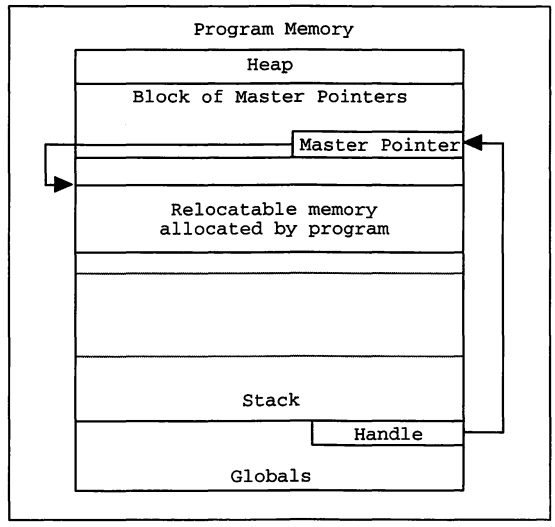


Figure 2.3 The relationship between handles, master pointers, and relocatable blocks of memory.

This brings up an interesting point. The master pointers can't move because if they did, your handles wouldn't be pointing to them anymore. For this reason, master pointers are fixed in memory. In order to reduce any fragmentation this might cause (you don't want to be creating and releasing master pointers every time a relocatable block of memory is allocated), a block of master pointers is allocated on the heap when your program runs. This fixed block contains 64 master pointers that your program can use. What if you need more? Whenever your program allocates that 65th block of memory, a new master pointer is automatically created. This

in itself could cause memory fragmentation, because the newly allocated master pointer is fixed in memory and won't move during heap compaction. Most times, though, this shouldn't be a problem because you'll probably not need more than 64 handles for your game. However, if you are certain you'll need more, you can manually create additional blocks of master pointers during your program initialization so that those addition blocks will remain near the top of the heap (and not get "in the way" later).

The following routines allow you to allocate, manipulate, and free handles to memory:

```
// Allocate a relocatable block of memory.  The new handle
// is unlocked and unpurgable (see handle flags below).
Handle NewHandle( Size sizeByteCount );

// Allocates a new master pointer, sets it to nil and returns
// a pointer to it.
Handle NewEmptyHandle( void );

// Returns a handle, given it's master pointer.
Handle RecoverHandle( Ptr ptrMaster );

// Free a relocatable block of memory.
DisposHandle( Handle handle );

// Get the size of an allocated relocatable block of memory.
Size GetHandleSize( Handle handle );

// Set the size of an allocated relocatable block of memory,
// which can be greater or less than the current size.
void SetHandleSize(   Handle      handle,
                      Size        sizeNew );

// Move a relocatable block of memory as near as possible to
// the end of the heap.  If you are locking a block of memory
// for an extended amount of time, using this call will
// minimize interference of the locked block with heap
// compaction.
void MoveHHi( Handle handle );

// Create a copy of a relocatable block of memory.  On input,
// the handle points to the source block.  On output, the
// handle points to the copy.
OSErr HandToHand( Handle *pHandle );
```

```
// Copy a fixed block of memory to a new relocatable block
// of memory.
OSErr PtrToHand(      Ptr         ptrSource,
                      Handle      hNewHandle,
                      long        lByteCount );

// Copy a fixed block of memory to an existing relocatable
// block of memory.
OSErr PtrToXHand(     Ptr         ptrSource,
                      Handle      handleDestination,
                      long        lByteCount );

// Append a relocatable block of memory after another.
OSErr HandAndHand(    Handle      hSource,
                      Handle      hDestination );

// Append a fixed block of memory after a relocatable block
// of memory.
OSErr PtrAndHand(     Ptr         ptrSource,
                      Handle      hDestination,
                      long        lByteCount );

// Purges a relocatable block of memory, but keeps it's
// master pointer which gets set to nil.  In most cases
// you'll use DisposHandle() to free a handle and it's
// master pointer.
void EmptyHandle( Hande handle );

// Reallocates a relocatable block of memory after it's
// handle has been purged using the EmptyHandle() call.
void ReallocHandle(   Handle      handle,
                      Size        sizeNeeded );

// Create an additional block of 64 master pointers.
void MoreMasters( void );
```

Handle Flags

Handles have three characteristics: *locked*, *purgable*, and *resource*. The locked flag determines whether or not a block of memory can be moved. A locked block of memory cannot be relocated. A purgable block of memory can be removed from the heap during a memory purge. The resource flag specifies whether or not a block of memory is a resource.

For most cases, the only flag you will need to change is the locked flag (using the `HLock()` and `HUnlock()` toolbox calls). The resource flag is automatically set for loaded resources, and the purgable flag is mainly a characteristic of resources. Of course, you could set your own allocated handle to be purgable, but that almost never makes much sense. You wouldn't want the Macintosh operating system to purge a block of memory you allocated when it feels like it because you might just try to use it later. If you didn't need the memory anymore, you'd have freed it yourself right?. Here are the routines to set and reset the resource and purgable flags of a handle (the locked flag will be discussed in detail in the next section):

```
// Make a relocatable block of memory purgable.
void HPurge( Handle handle );

// Make a relocatable block of memory unpurgable.
void HNoPurge( Handle handle );

// Set the resource flag of a handle.
void HSetRBit( Handle handle );

// Clear the resource flag of a handle.
void HClrRBit( Handle handle );
```

Locking Handles

One drawback to using handles is that it takes two dereferences to get at the data to which they point. First, you need to dereference the handle to get the master pointer, and then the master pointer to get at the data. If you are only doing this a couple of times, the time penalty of two dereferences probably won't hurt. However, speed is of the essence frequently in games. You may need to reference the data many times in a loop, and two dereferences are going to slow things way down. No, the solution is not to allocate a fixed block of memory when you'll need to access in a loop. Fixed memory is bad, remember. You could store a copy of the master pointer in a local variable and use it for the loop, eliminating the need to dereference the handle each time. This sounds like a reasonable solution until you think about the possibility that the memory block you're refer-

encing might move before you're through with it. The copy of the master pointer would then point to lala land, and a system error would be likely. So what's a poor programmer to do? Well, maybe if we could make the memory block fixed for a short period, while it's being processed, and then set it back to relocatable when we're done, all our problems would be solved. There are toolbox calls to do just this. So, when you need to process a block of relocatable memory (one to which you have defined a handle), you should lock it first, do your work, then unlock it so it can be moved later if necessary. Here are the routines:

```
// Lock a relocatable block of memory so it can't be moved.
void HLock( Handle handle );

// Unlock a relocatable block of memory so that it can be
// moved again.
void HUnlock( Handle handle );
```

The following code fragments show how to allocate a new memory block using a handle, and lock that handle so that it can be accessed quickly in a loop (the functions and definitions can be found in their entirety in **Scores Window.c** and **Scores Window.h**):

```
// These are fragments of the Desert Trek definitions, used for
// this example only.
typedef struct _SCORE
{
  long     lScore;
} SCORE, *PSCORE, **HSCORE;

typedef struct _SCORES
{
  SCORE  Score[10][10];
} SCORES, *PSCORES, **HSCORES;

static HSCORES  hScores;

void InitializeScoresWindow( void )
{
  short    sLoop;
  PSCORES  pScores;
```

```
    // Create a relocatable block of memory for the high scores
    hScores = (HSCORES) NewHandle( sizeof( SCORES ) );

    // Lock the block so it can't be moved while we're processing
    // it
    HLock( (Handle) hScores );

    // Get a copy of the master pointer
    pScores = *hScores;

    // Process the high scores data structure
    for( sLoop = 0; sLoop < 10; sLoop++ )
      pScores->Score[0][sLoop].lScore = 0;

    // Make the block movable again
    HUnlock( (Handle) hScores );
}
```

Referencing a Data Element Using a Handle

Here's a quick example of how to get at a data element of a handle pointing to a relocatable data structure. First you need to dereference the handle to get the master pointer, and then access the data element within the structure pointed to by the master pointer.

```
#define NAME_LENGTH 20

typedef struct _NAME
{
  char  szName[NAME_LENGTH + 1];
} NAME, *PNAME, **HNAME;

static HNAME hScoreName;

// Reference the szName element of the data structure pointed
// to by the hScoreName handle.
(*hScoreName)->szName
```

Memory Management Routines

You can let the Macintosh operating system take care of all the memory management for you, but if you would like to take a little control over

some of the memory yourself, the memory manager provides a number of toolbox calls:

```
// Returns the total number of bytes available in the heap.
long FreeMem( void );

// Returns the size of the largest block of memory that can
// be allocated via compaction.  No compaction actually takes
// place.
long MaxBlock( void );

// Obtains both the total number of bytes available and the
// largest contiguous block available on the heap if a purge
// occurred.  No purge actually takes place.
void PurgeSpace(  long *lTotalBytes,
                  long *lContiguousBytes );

// Compacts memory in an attempt to make a contiguous block
// of the size specified.  It returns the size of the largest
// block available after heap compaction.  No memory is
// allocated.
Size CompactMem( Size sizeNeeded );

// Purges all relocatable memory blocks that are purgable
// and unlocked in an attempt to make a contiguous block of
// the size specified available.  No memory is allocated.
void PurgeMem( Size sizeNeeded );

// Purges all purgable memory blocks, and compacts the heap
// in order to create the largest possible contiguous block
// of memory.  The function returns the size of the
// largest contiguous free block of memory.  The
// sizeExapandBytes parameter will contain the size by
// which the heap can be expanded.  No memory is actually
// allocated, and the heap is not expanded.
Size MaxMem( Size *sizeExpandBytes );

// Reserves a block of the requested number of bytes as
// near as possible to the start of the heap.  Compaction
// and purging may take place.  No memory is allocated.
void ResrvMem( Size sizeNeeded );

// Returns how much the stack can grow before colliding
// with the heap.
long StackSpace( void );
```

```
// Obtain a pointer to the end of the heap.
Ptr GetApplLimit( void );

// Set the heap limit to the specified address (not that
// this is not the new size of the heap, but an actual
// address).
void SetApplLimit( Ptr ptrNewLimit );

// Expands the heap to it's largest size, as defined by
// the current heap limit.
void MaxApplZone( void );
```

Strings

Remember, most toolbox calls that take strings as parameters require those strings to be in the Pascal format (with a length byte and no need for a null terminator). However, many times you'll want to use the standard C string functions to manipulate strings in your program. What's a poor programmer to do? Well, first of all, be very careful. You do not want to pass a C string to a toolbox call expecting a Pascal string. Neither do you want to pass a Pascal string to a C library call that's expecting a null-terminated C string. In both cases, you will get unexpected results (often leading to a system error).

Many compilers provide two functions that convert C and Pascal strings to the other. The converted string overlays the original string in memory (the two strings occupy up the same amount of storage—the Pascal string has a length byte and the C string has a null-terminating character).

```
// The first byte is the length indicator, so the maximum
// string length is 255 characters.
typedef unsigned char Str255[256];
typedef unsigned char *StringPtr;
typedef unsigned char **StringHandle;

unsigned char *CtoPstr(char *);
char *PtoCstr(unsigned char *);
```

One thing to keep in mind is that C strings are typically defined as `char[]`, whereas the C types for Pascal strings (`Str255`, `StringPtr`, and `StringHandle`) are defined as `unsigned char[]`. This means that you will need to typecast converted strings when making toolbox and C library calls (converted C strings will need to be typecast as (`unsigned char *`) when calling toolbox routines, and converted Pascal strings will need to be typecast as (`char *`) when calling C routines).

For Desert Trek, I have created two defines to make the typecasting a little more readable:

```
#define Pstr unsigned char *
#define Cstr char *
```

One final note. If you want to create a Pascal string constant, you need to prefix the string with '\p' or '\P'. In other words, the C string "Hello" would look like this: "\pHello". How about an example? The following fragment comes from the Desert Trek source file **File I/O.c.** Don't worry too much about the toolbox calls we haven't covered yet. They will be covered later.

```
#define FILE_ERROR_STRINGS        129
#define FILE_ERROR_PREFIX           1

#define Pstr unsigned char *
#define Cstr char *

static void PostIOError( OSErr osErr )
{
  Str255  str255;
  char    szError[80];

  // Loads a string from the resource file into str255.  This is
  // a Pascal format string.
  GetIndString( str255, FILE_ERROR_STRINGS, FILE_ERROR_PREFIX );

  // Convert the Pascal format string into a C format string.
  PtoCstr( str255 );
```

```
    // Copy the loaded string into the szError (both strings are C
    // format strings, but str255 needs to be typecast since it was
    // originally defined as a Pascal format string).
    strcpy( szError, (Cstr) str255 );

    // Convert the error number to a string.  str255 will now be
    // a Pascal format string again.
    NumToString( osErr, str255 );

    // Convert str255 to a C format string.
    PtoCstr( str255 );

    // Add the error number to the end of szError.
    strcat( szError, (Cstr) str255 );

    // Convert szError to a Pascal format string since we want to
    // use it in a toolbox call.
    CtoPstr( szError );

    // Set szError as a parameter string to a dialog.  Remember,
    // toolbox calls require Pascal format strings.  Notice that
    // the empty strings are specified in Pascal format ("\p").
    ParamText( (Pstr) szError, "\p", "\p", "\p" );
}
```

String/Number Conversion Toolbox Routines

It is frequently handy to convert numbers to strings and strings to numbers. For example, you may want to display a number on the screen, which requires a string representation of that number. There are, of course, standard C library calls to do this for you, but the Macintosh toolbox also provides a couple of calls. Of course, the strings used as input and returned as output are Pascal format strings. This is exactly why you might want to use these calls. Since you are probably going to get the strings from a toolbox call, you might as well convert them to a number without having to first convert the string to a C format string (which you would need to do if you used a C library call to convert the string to a number).

```
// Converts a long integer into a Pascal format string.
void NumToString(  long      lInput,
                   Str255    str255Output );

// Converts a Pascal format string to a long integer.
void StringToNum(  Str255    str255Input,
                   long      *lOutput );
```

Events

The main task of any Macintosh application is to receive and process events. This is especially true for games because the player drives what happens in the game. In addition to responding to events, a game usually needs to do some background processing even when the user generates no events. In other words, you may need to move an enemy ship, play a sound, or update a timer even if the player doesn't enter a move. Lastly, your game isn't the only thing running on the Macintosh. Good Macintosh applications, including games, occasionally give up some time to the system so that other processes can get the CPU (which includes the Macintosh operating system itself). Do all this and have great graphics. Sounds tough to accomplish, doesn't it? Don't worry, though, it's easy once you know the trick.

Waiting for and Getting Events

There are two ways to check for events posted to your game. The original method used to check for events required you to repeatedly call a toolbox function to see if there was an event posted to your application. If no event was posted, you needed to continue calling the function until you finally got an event. This method also required you to call a different toolbox function to give up CPU time to the system. After System 7 was released, a new toolbox call was added that allows you to wait for events. You no longer need to repeatedly call a function, since this toolbox call doesn't return until an event is posted to your application. Also, while

you're waiting, the system gets all the CPU time it needs. This new toolbox call also allows you to specify a timeout value so that you can do any background processing your game needs. However, you can't assume that the Macintosh your game is running on has the new toolbox call, so you must be prepared to use either method.

To determine if the Macintosh you're game is running on supports a particular toolbox call, you can use the following toolbox function:

```
// TrapType values
enum { OSTrap, ToolTrap };

//  Get an operating system or toolbox trap address.
long NGetTrapAddress(  short     sTrapNumber,
                       TrapType  trapType );
```

To check if a particular toolbox call exists, you need to get its trap address and compare it to something. That something is the address of a special trap, the `_Unimplemented` trap. So, if the trap address of the toolbox call you're inquiring about has the same address as the `_Unimplemented` trap's address, the Macintosh your game is running on does **not** support the toolbox call in question. So, how do you know the trap number of a particular toolbox call? In general, it's the toolbox call prefixed with an underscore (for example, `_WaitNextEvent`). Your C compiler should come with a header file, Traps.h, that lists all the trap numbers.

The following code sample shows how to use `NGetTrapAddress()` to determine whether or not the Macintosh supports the `WaitNextEvent()` toolbox call:

```
static Boolean bHasWaitNextEvent = false;

// This Mac supports WaitNextEvent if NGetTrapAddress doesn't return the
// _Unimplemented trap address.
bHasWaitNextEvent = (      NGetTrapAddress( _WaitNextEvent, ToolTrap ) !=
                           NGetTrapAddress( _Unimplemented, ToolTrap ) );
```

Now that you know whether this Macintosh supports the `WaitNextEvent()` toolbox call, here's what it looks like:

```
// Event Masks for WaitNextEvent() and GetNextEvent()
enum {
everyEvent = -1,           // Check for all events
mDownMask = 2,             // Mouse down events
mUpMask = 4,               // Mouse up events
keyDownMask = 8,           // Key down events
keyUpMask = 16,            // Key up events
autoKeyMask = 32,          // Key repeat events
updateMask = 64,           // Update events
diskMask = 128,            // Disk events
activMask = 256,           // Activate events
highLevelEventMask = 1024, // High level events
osMask = -32768            // Operating system events
};

// The Event Record Structure
struct EventRecord
{
  short  what;       // Event type
  long   message;    // Event specific data
  long   when;       // Time when event occurred
  Point  where;      // Mouse location of event
  short  modifiers;  // Keyboard and Mouse state
};

// Get's the next event posted to your application.  If
// there's no event, let other processes use the CPU for
// up to the amount of time specified in lSleepTime.
Boolean WaitNextEvent(  short          sEventMask,
                        EventRecord    *pEvent,
                        long           lSleepTime,
                        RgnHandle      hRgnMouse );
```

In a moment, we'll look at the `EventRecord` structure in more detail. First, though let's examine the `WaitNextEvent()` parameters. The `sEventMask` parameter specifies the events we want to look for. Typically, you'll want to look for all events posted to your game. Note that even if you ignore certain events by masking them out, those events remain on your game's event queue. Eventually, you'll need to process them. The `EventRecord` structure returns the details of the event posted to your game. Again, more on that later. The `lSleepTime` parameter specifies how long you're willing to wait for an event without getting any CPU

time to do background processing for your game. If your game has no real-time components, you could wait indefinitely. However, you usually will want to do something about once every clock tick. (By the way, a clock tick is 1/60th of a second.) The `hRgnMouse` specifies a region (regions will be discussed in Chapter 7 on Quickdraw) in which your application will get mouse move events. Most of the time, you won't need to worry about mouse move events, so you'd pass `nil` as this parameter. Finally, this routine returns `true` or `false`, depending on whether an event was posted to your game (in other words, `true` means you got an event, `false` means that you didn't get an event and the sleep time period elapsed).

If the Macintosh your game is running on does not support `WaitNextEvent()`, you need to use the following toolbox calls:

```
// Get the next event for your application from the event
// queue (if there is an event).
Boolean GetNextEvent(  short        sEventMask,
                       EventRecord  *pEvent );

// Give up the CPU to other processes.  Apple recommends that
// it be called at least once a tick (1/60th of a second).
// You should call it just before GetNextEvent().  Even if
// you are using WaitNextEvent(), you need to use this call
// if your game does a lot of processing based on an event so
// that other tasks don't get "starved" while you do your
// processing.
void SystemTask( void );
```

Previously in this chapter, you saw the `main()` function of Desert Trek. Basically, all it did was call another Desert Trek function to check for events until the program ended. Here's how Desert Trek checks for events:

```
static void CheckEvent( void )
{
  EventRecord  eventRecord;
  Boolean      bEvent;
  short        sItemHit;
  DialogPtr    pDialog;

  // If this Macintosh supports WaitNextEvent(), use it to wait
```

```
  // for an event.  We'll wait for up to 1 tick for an event
  // before dropping out so we can do background processing.
  if ( bHasWaitNextEvent )
    bEvent = WaitNextEvent( everyEvent, &eventRecord, 1L, NULL );

  // This Mac doesn't support WaitNextEvent(), so we need to use
  // GetNextEvent().  In addition, we need to give the system
  // some time to process other applications, so we also need to
  // call SystemTask().
  else
  {
    SystemTask();
    bEvent = GetNextEvent( everyEvent, &eventRecord );
  }

  // Perform any background processing.  Desert Trek doesn't
  // really have any other than to check to see if any sounds
  // that were playing are finished.
  CheckSound();

  // If there was an event, process it.
  if ( bEvent )
    HandleEvent( &eventRecord );

  // There was no event, but since Desert Trek uses modeless
  // dialog boxes, we need to check to see if there was a
  // dialog event and if so, call DialogSelect() to make sure
  // that text edit cursors blink correctly in that dialog.
  else
    if ( IsDialogEvent( &eventRecord ) )
      DialogSelect( &eventRecord, &pDialog, &sItemHit );
}
```

Let's examine the `CheckEvent()` function in detail. Remember, there are two toolbox calls to get an event posted to your program. The preferred method is `WaitNextEvent()` because it automatically gives CPU time to other processes running on the Macintosh. However, if `WaitNextEvent()` is not supported, you'll have to use `GetNextEvent()` along with `SystemTask()` (in order to give up the CPU to other processes). In any case, both toolbox calls return `true` if an event was retrieved for your game, `false` if no event was retrieved. Before processing an event, if there was in fact one, Desert Trek calls its own function, `CheckSound()`. This is the only work Desert Trek per-

forms in the background. You'll want to put anything else your game does in the background here. When I say background, I mean any work your program has to do that is, for the most part, independent of events. This might be to move the asteroids a little, scroll an introduction story (as in my Galactic Empire game, which you can find on the CD-ROM), or close a no longer needed sound channel (which is what Desert Trek does).

The last thing `CheckEvent()` does is determine whether there was an event posted to Desert Trek. If an event was posted, the routine `HandleEvent()` is called to deal with it. If no event was posted, though, we aren't done. Desert Trek uses modeless dialog boxes. Modeless dialog boxes, which are described in much more detail in Chapter 6, are essentially special dialog boxes that can remain on the screen along with your other game windows. Many dialog boxes have text edit boxes within them to allow the user to input text (such as a name for the high scores list) and need to have the cursor within the selected text edit field blink. A special event is posted to your game to enable the cursor to blink within a modeless dialog boxes text edit field. The interesting thing about this event is that `WaitNextEvent()` and `GetNextEvent()` both return `false`, as if there wasn't really an event for your game. This is somewhat true, since your game didn't really get an event worth processing. The event was really meant for any modeless dialog boxes you have. In any case, even if you think no event was posted to your game, you need to call the `IsDialogEvent()` toolbox call to see if any dialogs your game displays have a special event waiting for them. If so, the `DialogSelect()` toolbox call automatically handles the special event destined to the dialog (so you don't need to bother with it). More on these toolbox calls is discussed in Chapter 6.

Determining What Event Occurred

After your game receives an event, what do you do with it? The first thing you need to do is determine the type of event. Was it mouse click, keystroke, or update event? The following is a list of events your game might receive, which gets reported to your game in the `what` field of the `EventRecord` data structure:

```
enum
{
  nullEvent = 0,        // Null event (no event available)
  mouseDown = 1,        // Mouse down event
  mouseUp = 2,          // Mouse up event
  keyDown = 3,          // Key down event
  keyUp = 4,            // Key up event
  autoKey = 5,          // Auto key (repeated key)
  updateEvt = 6,        // Update event
  diskEvt = 7,          // Disk event
  activateEvt = 8,      // Activate event
  osEvt = 15,           // Operating system event
};
```

Desert Trek cares about the following events: `mouseDown`, `keyDown`, `autoKey`, `udpateEvt`, `activateEvt`, and `osEvent`. For most cases, these are the only events your game need to respond to. Here's the `HandleEvent()` routine from Desert Trek:

```
static void HandleEvent( EventRecord *pEvent )
{
  switch ( pEvent->what )
  {
    case mouseDown:

      HandleMouseEvent( pEvent );
      break;

    case keyDown:
    case autoKey:

      HandleKeyEvent( pEvent );
      break;

    case updateEvt:

      HandleUpdateEvent( pEvent );
      break;

    case activateEvt:

      HandleActivateEvent( pEvent );
      break;
```

```
    case osEvt:

      HandleOSEvent( pEvent );
      break;
  }
}
```

Handling Mouse Events

Mouse events are probably the most difficult to process since the mouse can be used to do many different things in an application. These actions include selecting a menu item, moving a window, or clicking on a button in a window or dialog box. The first thing you need to do is determine where the mouse event occurred. Windows are broken up into different regions, such as the title bar, close box, or the content of the window itself. You'll need to determine not only which window the mouse event affects, but also the region of that window where the mouse was clicked. The following toolbox call does just that:

```
short FindWindow(  Point       ptMouseLocation,
                   WindowPtr   *pWindow );
```

You simply supply the `where` field of the `EventRecord` data structure, and get back the `WindowPtr` of the window affected as well as the region within the window where the mouse was clicked (window pointers are discussed in Chapter 4.) This window region is commonly referred to as the window "part," and the following values represent the window parts that can be returned by `FindWindow()`:

```
enum
{
  inDesk = 0,          // Macintosh Desktop
  inMenuBar = 1,       // Menubar
  inSysWindow = 2,     // System window
  inContent = 3,       // Content region of a window
  inDrag = 4,          // Title bar
  inGrow = 5,          // Grow region
  inGoAway = 6,        // Close box
  inZoomIn = 7,        // Zoom region for a "zoomed-out" window
  inZoomOut = 8        // Zoom region for a "zoomed-in" window
};
```

If the mouse event occurs in a system window, you don't want to process it yourself. Instead, you'll want to pass that event back to the system using the following toolbox call:

```
void SystemClick( EventRecord   *pEvent,
                  WindowPtr     pWindow );
```

For mouse clicks in the zoom box, close box, size region, and title bar, all you need to do is call the following toolbox routines and the appropriate window action will be taken care of for you. These routines take two common parameters: `ptStart`, which specifies where the mouse was clicked, and `pWindow`, which specifies the window to affect. You need to pass the `where` parameter of the `EventRecord` data structure for `ptStart`, and the `pWindow` returned by `FindWindow()` for `pWindow`.

```
// Moves a window based on a user click event in the window's
// title bar.  rectDragLimit limits where the user can move
// the window.  Basically, it's where the user is allowed to
// release the mouse.
void DragWindow(     WindowPtr     pWindow,
                     Point         ptStart,
                     Rect          rectDragLimit );

// Automatically sizes the window, based on mouse movement
// by the user.  You need to specify the maximum size of the
// window in the rectSizeLimit parameter.
long GrowWindow(     WindowPtr     pWindow,
                     Point         ptStart,
                     Rect          rectSizeLimit );

// Tracks the mouse in the zoom box of a window.  You need
// to pass the window part returned by FindWindow() as the
// sWindowPart parameter.  This call returns true if the
// mouse button is released in the zoom box, false if not.
// If this toolbox call returns true, you need to call
// ZoomWindow() to actually perform the zoom.  See chapter 4
// on windows for the prototype of ZoomWindow().
Boolean TrackBox(    WindowPtr     pWindow,
                     Point         ptStart,
                     short         sWindowPart );

// Tracks the mouse in the close box of a window.  This call
// returns true if the mouse button is released in the
```

```
// close box, false if not.  If this toolbox call returns
// true, you need to close the window yourself (typically
// with CloseWindow(), which is discussed in chapter 4).
Boolean TrackGoAway( WindowPtr     pWindow,
                     Point         ptStart );
```

The following routine from Desert Trek handles mouse down events:

```
static void HandleMouseEvent( EventRecord *pEvent )
{
  short       sWindowPart;
  short       sWindowID;
  WindowPtr   pWindow = nil;
  Rect        rectDragArea;

  // Find in which window and window part the mouse was clicked.
  sWindowPart= FindWindow( pEvent->where, &pWindow );

  // Get the Desert Trek window ID of the window clicked.  More
  // on this can be found in chapter 4 on windows.
  sWindowID = (short) GetWRefCon( pWindow );

  // If the user clicked in a system window, let SystemClick()
  // handle it.
  if ( sWindowPart == inSysWindow )
    SystemClick( pEvent, pWindow );

  // If the user clicked in the menubar, see which menu item was
  // selected.  More on this can be found in chapter 5 on menus.
  else if ( sWindowPart == inMenuBar )
    HandleMenuSelection( MenuSelect( pEvent->where ) );

  // If an application modal dialog is being displayed, and the
  // window the user clicked on wasn't the dialog, beep since the
  // user is not allowed to work with other Desert Trek windows
  // when an application modal dialog is up.  See chapter 6 for
  // more information.
  else if ( ( IsAppModalDialogUp() ) &&
            ( pWindow != GetAppModalDialogWindow() ) )
    SysBeep( 1 );

  // If the Desert Trek window the user clicked on wasn't the
  // frontmost window, make it the frontmost window.
  else if ( pWindow != FrontWindow() )
```

```
      SelectWindow( pWindow );

    // See what part of the window the user clicked on.
    else switch ( sWindowPart )
    {
      // User wants to move the window.
      case inDrag:

        // Let the user move the window anywhere on the screen.
        rectDragArea = screenBits.bounds;
        DragWindow( pWindow, pEvent->where, &rectDragArea );
        break;

      // The user might be trying to close the window.
      case inGoAway:

        if ( TrackGoAway( pWindow, pEvent->where ) )
          HandleCloseWindow( sWindowID );
        break;

      // The user clicked in the content of a Desert Trek window.
      case inContent:

        HandleMouseDownInContent( pEvent, sWindowID );
        break;
    }
}
```

Handling mouse down events in the content region of a window is something many games are going to need to do, and we'll take a closer look at processing them in Chapter 4.

Handling Keyboard Events

When the user presses a key, the game will receive a keyboard event. There are two basic types of keyboard events, but usually you will handle them in the same way. The first is the `keyDown` event, which occurs when the user presses a key. The second is the `autoKey` event, which occurs when the user holds down a key for a certain amount of time and the key starts automatically repeating (so, you'll first get the `keyDown` event, then a little bit later, many `autoKey` events until the user releases the key).

In addition to the key being pressed, you frequently need to know what modifier keys were held down. For example, if a key press is accompanied by the **Command** key, chances are that the user is attempting to use a **Command** key shortcut for a menu item. Modifier keys include the **Shift**, **Command**, **Control**, and **Option** keys, as well as whether the **Caps Lock** key was down or not. The state of the modifiers can be found in the `modifiers` field of the `EventRecord` data structure. You need to use bit operators to determine which of the following modifier keys were pressed for the event. A code sample showing how to do this follows:

```
enum
{
  btnState = 128,        // Mouse button down?
  cmdKey = 256,          // Command key down?
  shiftKey = 512,        // Shift key down?
  alphaLock = 1024,      // Caps lock down?
  optionKey = 2048,      // Option key down?
  controlKey = 4096      // Control key down?
};
```

The `message` field of the `EventRecord` data structure contains the key pressed. You can extract either the character code or key code from this field using the following two masks:

```
enum
{
  charCodeMask = 0x000000FF,    // The character itself
  keyCodeMask = 0x0000FF00      // The key code
};
```

The following code segment shows how Desert Trek handles a key event. Notice that the code first determines which window the keystroke belongs to by using the `FrontWindow()` toolbox call (which is described in Chapter 4). Each type of window in Desert Trek (the main window, the high scores window, the about window, etc.) has its own event-handling routine, and the correct one needs to be called when a key event is posted.

```
static void HandleKeyEvent( EventRecord *pEvent )
{
  short      sWindowID;
  WindowPtr  pWindow = nil;

  // Determine which window was the front window - it gets the keystroke.
  // See chapter 4 on Windows for a description of these toolbox calls.
  pWindow = FrontWindow();
  sWindowID = (short) GetWRefCon( pWindow );

  // If the command key was down, handle it as a menu selection.  See
  // chapter 5 on menus for the MenuKey() toolbox call.  Notice that
  // we pass the character code to the MenuKey() function.
  if ( pEvent->modifiers & cmdKey )
    HandleMenuSelection( MenuKey( (char) ( pEvent->message &
                         charCodeMask ) ) );

  // Pass the key event to the appropriate routine.
  else switch ( sWindowID )
  {
    case HELP_WINDOW_ID:

      DoInfoWindowEvent( GetInfoWindowPtr( HELP_WINDOW_ID ), pEvent );
      break;

    case CARYS_GAMES_WINDOW_ID:

      DoInfoWindowEvent( GetInfoWindowPtr( CARYS_GAMES_WINDOW_ID ),
                         pEvent );
      break;

    case SCORES_WINDOW_ID:

      DoScoresWindowEvent( pEvent );
      break;

    case ABOUT_WINDOW_ID:

      HandleAboutWindowEvent( pEvent );
      break;

    case APP_MODAL_DIALOG_ID:

      DoAppModalEvent( pEvent );
      break;
  }
}
```

The following code segment shows how to determine which key was pressed (the complete function's code can be found in **Scores Window.c**).

```
void  DoScoresWindowEvent( EventRecord  *pEvent )
{
  short       scharCode;

  // If this is a key down event
  if ( pEvent->what == keyDown )
  {
    // Get the character pressed using the BitAnd() toolbox call.
    // The BitAnd() toolbox call is equivalent to the & operator.
    // e.g.:  scharCode = pEvent->message & charCodeMask
    scharCode = BitAnd( pEvent->message, charCodeMask );

    // If the Return or Enter key was pressed, simulate a click
    // on the OK button.
    if ( ( scharCode == 13 ) || ( scharCode == 3 ) )
    {
      // Highlight the OK button and close the high scores window.
    }

    // If 0 through 9 was pressed, change the skill level being
    // viewed in the high scores screen (0 means skill level 10).
    else if ( ( scharCode >= '0' ) && ( scharCode <= '9' ) )
    {
      // Change the skill level viewed.
    }
  }
}
```

Handling Update Events

Update events occur when a window needs to be redrawn. For reasons why a window needs to be redrawn, see Chapter 4. The most important thing to do here is to determine which window needs to be updated. The `message` field of the `EventRecord` structure contains the window pointer of the window that needs updating. The following code snippet shows how to get the window pointer from the event:

```
static void HandleUpdateEvent( EventRecord *pEvent )
{
  WindowPtr  pWindow = nil;

  pWindow = (WindowPtr) pEvent->message;
}
```

Update events will be covered in much more detail in Chapter 4.

Handling Activate Events

Activate events occur when one of your game windows becomes active or inactive. When the user clicks from window to window in your game, the new window clicked on gets an activate event while the window that used to be active gets a deactivate event. For the most part, you game needs to do very little in response to an activate event. However, if the window getting activated or deactivated contains a scrollbar, you need to active or deactivate that scrollbar accordingly. To determine which window in your game received the activate or deactivate event, simply use the message field of the event record, just as in the case of the update event. Lastly, you need to know whether or not the specified window was activated or deactivated. This information is contained within the modifiers field of the event record. The toolbox defines a bit mask that allows you get at the modifier bit for activate events, called activateFlag. The following function from Desert Trek handles an activate event, determining which window the event was for, and whether the event was an activate or deactivate event. This function can be found in "Main.c"

```
static void HandleActivateEvent( EventRecord *pEvent )
{
  WindowPtr  pWindow = nil;
  Boolean    bActivated;

  // Determine the window that got the activate event.
  pWindow = (WindowPtr) pEvent->message;

  // Determine whether the window was activated or deactivated.
  bActivated = (Boolean) BitAnd( pEvent->modifiers, activeFlag );
```

```
    // Process the activate event.
    ActivateDeactivateWindow( pWindow, bActivated );
}
```

Handle Operating System Events

The operating system events that you would want your game to respond to include suspend and resume events. A suspend event is sent to your game when the user switches to another application running on the Mactinsoh. A resume event gets sent to your game when the user switches from another application running on the Macintosh to your game. Chapter 10 on Sound will cover these events in detail since you'll primarily need to take care of sound chores when receiving one of these operating system events.

General Event Toolbox Routines

There are several additional even-trelated toolbox calls that you may find useful. Here is a list of them:

```
// Checks for an event without removing that event from the
// event queue.  Returns true if an event is found, and
// pEvent contains the event record data for the event.
Boolean EventAvail( short          sEventMask,
                    EventRecord    *pEvent );

// Removes all events of type sEventMask from the event
// queue.  sStopMask causes FlushEvents() to stop when an
// event of that type is encountered (in other words, all
// events in the queue after sStopMask are kept, even if they
// are of type sEventMask).
void FlushEvents(   short          sEventMask,
                    short          sStopMask );

// Get the current mouse location in local coordinates.  See
// chapter 7 on quickdraw for the Point definition as well as
// the difference between local and global coordinates.
void GetMouse( Point &ptMouseLocation );
```

```
// Returns true if the mouse button is down.
Boolean Button( void );

// Returns true if the mouse is down and there are no mouse
// events pending in the event queue.  In other words, the
// mouse is still down from the previous mouse down event.
Boolean StillDown( void );

// Does the same as StillDown(), but removes the
// corresponding mouse up event from the event queue before
// returning false.
Boolean WaitMouseUp( void );

// Returns a bit array of which keys are currently pressed
// on the keyboard.  At first, this seems like a good way to
// get keys for a game, since you can just read the keyboard
// state at any time.  However, note that only two keys will
// be reported down by the Macintosh at any one time - it's a
// limit imposed by the Macintosh (this doesn't include the
// modifier keys, so you get up to two keys and the state of
// the shift, caps lock, control, option, and command keys).
typedef long KeyMap[4];
void GetKeys( KeyMap keyMapKeys );

// Returns the number of ticks since the system was started.
// A tick is approximately 1/60th of a second.
long TickCount( void );

// Causes your program to stop for lDuration ticks.  It's
// primary use in games is to cause animation rates to be
// the same on different speed Macs (you can slow down faster
// Macs so the game isn't too fast).  However, you should
// never use this function since it causes the entire Mac to
// come to a screeching halt, and that's poor form in a
// cooperative multitasking environment.  The same effects
// can be achieved using another method that allows the Mac
// to continue processing other applications.  Chapter 7 on
// quickdraw covers the other method.  I show you this
// toolbox call so that I can tell you never, ever, under any
// circumstances to use it!!!!
void Delay(  long  lDuration,
             long  *plEndTickCount );
```


TIP

In case you missed the point, you should never use the Delay() toolbox call in your game. However, there are frequent times when you'd like to have events take the same amount of time regardless of how fast the Macintosh your game is running on can execute your code. For example, Desert Trek's scene fades take the same amount of time on any Macintosh, regardless of its speed (well, if you want to get really technical, super slow Macintoshes might cause the fade to take a little longer, but I won't admit to it!). If I did not put some type of delay in there, very fast Macintoshes would execute the scene fade code so fast that the player would completely miss it. How did I do it? Well, I created a routine called NiceDelay() which delays for the specified amount of time (just like the Delay() toolbox routine), but gives other tasks running on the Macintosh processor time. Here's the code, which can be found in "Common Functions.c".

```
void NiceDelay( long lDelay )
{
  long  lStartTime;
  long  lEndTime;

  // Initialize the timer.
  lStartTime = TickCount();

  // Delay for the specified amount of time, giving other processes the chance
  // to execute while we're delaying.  Notice that we continue to perform
  // Desert Trek's background processing while delaying.  The only background
  // processing Desert Trek does is to check to see if any sounds have
  // finished playing.
  do
  {
    SystemTask();
    CheckSound();
    lEndTime = TickCount();
  } while( ( lEndTime - lStartTime ) < lDelay );
}
```

Random Numbers

Just about every game needs to generate random numbers to drive game events. The Macintosh toolbox provides two routines that can be used to generate random numbers. Before you generate random numbers,

though, you need to understand the concept of a seed. You see, the Macintosh doesn't truly generate completely random numbers. Random numbers are really computed, based on a seed number. The sequence of random numbers generated for the same seed are always the same. When your game starts, it is given the same seed number each time. This means that the same sequence of random numbers will be generated each time your game runs. This, of course, probably isn't at all what you want. You want a different sequence of random numbers to be used each time your game runs. Otherwise, you could predict what would happen in your game (or worse yet, the user could predict the sequence of events for your game). To overcome this limitation, you should always set the seed number when your game begins. However, what should you set it to? A good method to use would be to set the seed value to the current time, or tick count. This would pretty much ensure a different seed everytime your game is run. The seed is stored in a global variable called randSeed. It is defined by the Macintosh as follows.

```
long randSeed;
```

The first thing your should do when your game starts is to set the seed number using the TickCount() toolbox funtion. The following example shows how to do this.

```
// Sets the random number seed.
randSeed = TickCount();
```

Now that the random number seed has been taken care of, how do you generate random numbers? The toolbox provides the following routine that generated random numbers between -32768 and +32767.

```
// Returns a random number between -32768 and +32767.
short Random( void );
```

This is just great, but most of the time you'd like to generate a random number between 0 and some value, like say 100. How can you use the Random() routine to generate a number between 0 and some limit that you define? You can simply use the modulo operator, which gives you the

remainder of a division operation. Thus, a random number modulo 100 gives you a number between 0 and 99 (which is a range of 100 values). The following code example generates a random number between 0 and 99. Notice that you must take the absolute value of the Random() function otherwise you'd get a range of -99 to +99.

```
short sMyRandom;

sMyRandom = abs( Random() ) % 100;
```

If you want to generate a number between 1 and 10, using the following code example.

```
sMyRandom = 1 + abs( Random() ) % 10;
```

CHAPTER 3

RESOURCES

All Macintosh files consist of two forks, one for the salad and one for the main course. Proper etiquette must be observed, otherwise your friends may stop asking you out to dinner. Seriously, Macintosh files do consist of two forks, the data fork and resource fork. Each fork is independent, meaning file I/O to one fork does not affect the other, and a Macintosh file may contain either a data fork, resource fork, or both.

Typically, data files contain only a data fork, and native 680x0 application files (the actual executable program itself) contain only a resource fork. Native 680x0 program code is contained in the resource fork of type CODE. Native PowerMac code is stored in the data fork. This scheme for storing program code makes it possible for a single Macintosh executable file to contain both native PowerMac code (in the data fork) and native 680x0 code (in the resource fork).

The Resource Fork

This chapter concerns itself with the resource fork of a file. For more details on the data fork, see Chapter 9. The resource fork of an application file contains definitions for the program's menus, dialog boxes, graphics, icons, cursors, text, sound, program specific data, and, to be honest, just about everything else. Proper use of the resource fork allows you to customize your game without needing to change the code. For example, let's say you want to support multiple languages for your program. If you store all program text in string resources, translating your program into a different language is as simple as changing those string resources. I had to learn that lesson the hard way, of course.

When I wrote *Galactic Empire*, all of the text used for screen drawing was hard coded into the program. I got a request from a user who wanted to translate the game into a different language so that his kids could play the game in their native tongue. He was able to translate the menu and dialog box text easily into the other language because they were resources easily edited by ResEdit, but he could not change the hard-coded strings. (For you technophiles out there, yes, he could have edited the 'CODE' resources containing the hard-coded strings, but what if the translated string was larger than the original?) If I had put all text strings in the resource file, his job would have been easy. However, I needed to change the code to read all strings from the resource file before the translation could be completed.

Creating the resources for your game will take at least as long as it takes to write the code. This will become especially true when you have a little more coding experience and can re-use code from other projects. Of course, you'll want to start coding before creating all the resources for your game, but it is often necessary to create many of the resources first. For example, you'll need menu and window resources before you can write even a minimally functional program.

The remainder of this chapter discusses the creation of various types of resources using ResEdit, the standard Macintosh resource editor. At the end of this chapter, I will demonstrate the power of resources by

transforming Desert Trek into the exciting new game, Vegas Trek, without rewriting a single line of code.

ResEdit

There are generally two ways to create and edit resources for your game. One such method is to use a resource compiler. A resource compiler takes an input file describing the resources to create in textual format, and generates a resource file as output. The resultant resource file can then be combined with the compiled code to produce the final application. Another, and by far, more popular method to create and edit resources is to use Apple Computer's ResEdit. ResEdit allows you to directly edit resource files in a graphical environment. It even allows you to edit an application program's resource fork. After the compiler itself, ResEdit will be your most valuable tool.

There is a certain *mystique* concerning the use of ResEdit in the user community. Often you are told to use it with care, and always, always use it on backups of programs, not the original itself. Of course, editing the system file with ResEdit is certain to quickly destroy your computer. This reputation should tell you two things. First, ResEdit is a powerful tool that allows one to edit the resource fork of a program, significantly altering the way it operates. Second, we developers want to keep this powerful tool out of the hands of the masses so that we can demonstrate our incredible Macintosh knowledge time and again to impress our friends.

As a developer, you can, for the most part, ignore these warnings. However, they didn't arise simply to discourage others from modifying your programs. ResEdit can wreak severe damage to your resource files under certain circumstances. I strongly urge you to back up your resource files frequently, because if ResEdit encounters any problems such as lack of memory to perform an operation, it has a tendency to render your entire resource file completely and utterly useless. I've had this happen to me in the past, and I've learned my lesson the hard way—losing forever dozens of hours of work. However, ResEdit is, for the most part, a very stable tool.

Using ResEdit

Though I can't possibly cover the use of ResEdit in full here, I can give you enough information to start using the tool productively. Again, ResEdit allows you to edit the resource fork of any Macintosh file. It does not allow you to view or edit the data fork.

Creating a Resource File

When you start a new project, one of the first things you'll need to do is create a resource file. If you are using Symantec C++ (or Think C), you should give your resource file the same name as your project file, with .rsrc appended to the end. For example, if the project file name is Desert Trek.π, name the resource file Desert Trek.π.rsrc. Using this naming convention allows your program, when run under the Symantec environment, to automatically load resources from your resource file. Otherwise, you'll need to explicitly add the resource file to your project, which will cause it to be compiled every time you change the resource file (which, believe me, will be often), increasing the time it takes to compile and run your program. In addition, you'll end up with a file of that name anyway (the *compiled* version of your resources), just begging to cause you endless confusion as to which resource file you should be editing anyway.

To create a resource file, start ResEdit. You'll be presented with the famous Jack-in-the-box opening screen. Click the mouse anywhere to continue. Next, you'll be presented with an Open File dialog box. At this point, select the resource file you want to edit, or click on the **New** button to create a new resource file. Use the naming convention described previously, and store the resource file in the same folder as your project. Make a backup of this file frequently as you'll be very, very upset if it ever gets corrupted.

Creating a Resource

Once in ResEdit, you'll see a window that is either blank for a new file or shows you all the resource types contained within the file. Since you can,

and must, have multiple resources of the same type (for example, two MENU resources to describe the File menu and Edit menu), the initial screen shows you just the types of resources in the file. To see all the resources of a particular type, simply double-click on the type and another window opens (see figure 3.1).

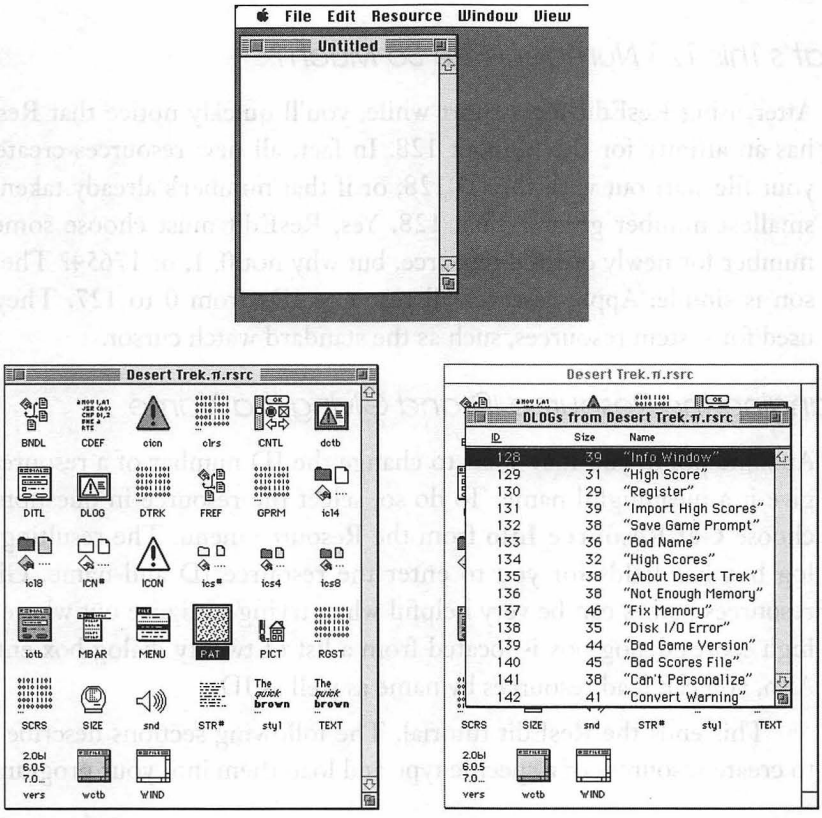


Figure 3.1 An empty resource file, a resource file with several resource groups, and a resource file with one resource group open.

To create a new resource, you need to use the **Create New Resource** command under the Resource menu (do not use the **New** command under the File menu because that gives you a new resource *file*). You can create new resources in one of two places: at the main window, which shows you all the resource types, or at any window that shows you all the

resources for a given type. Creating a new resource from the main window prompts you for the resource type to create. This is how you create a new resource type. Creating a resource from a window showing you all the resources of a particular type creates a resource of that type. Double-clicking on one of the existing resources displayed in this window will allow you to edit it.

What's This 128 Number I See So Much?

After using ResEdit for a short while, you'll quickly notice that ResEdit has an affinity for the number 128. In fact, all new resources created in your file start out with the ID 128, or if that number's already taken, the smallest number greater than 128. Yes, ResEdit must choose some ID number for newly created resource, but why not 0, 1, or 17654? The reason is simple: Apple reserves all resource IDs from 0 to 127. They are used for system resources, such as the standard watch cursor.

Changing the Resource ID and Giving It a Name

At some point, you may want to change the ID number of a resource, or give it a meaningful name. To do so, select the resource in question and choose **Get Resource Info** from the Resource menu. The resulting dialog box has fields for you to enter the resource ID and name. Giving resources names can be very helpful when trying to figure out where that high scores dialog box is located from a list of twenty dialog box entries. Also, you can load resources by name as well as ID.

This ends the ResEdit tutorial. The following sections describe how to create resources of a specific type and load them into your program.

'MENU' and 'MBAR' Resources

Every true Macintosh application, including games, has a menu bar associated with it. The definition of the menu bar and its menus are found in the 'MENU' and 'MBAR' resources.

Creating a 'MENU' Resource

The 'MENU' resource contains all information needed to describe a single Macintosh menu. You will need one 'MENU' resource for each menu in your game. ResEdit provides an easy to use 'MENU' editor that allows you to easily build and customize menus for your game.

Using your newfound ResEdit skills (or old ones if you're already a pro), create a 'MENU' resource. The 'MENU' dialog box shows you the menu you're building as well as options for each menu item (see Figure 3.2). A new menu has only a menu title, which you can change by typing in the Title edit field. Note that you can make this the Apple menu by selecting the radio button just under the Title edit field. Selecting this will create a menu with the Apple symbol as its title. You will need one of these for each game you write.

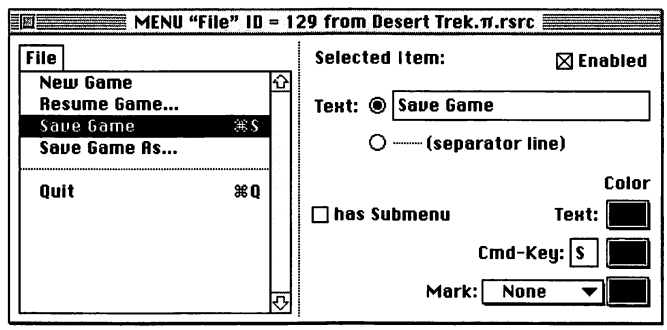


Figure 3.2 The Build Menu dialog box from ResEdit.

To add items to the menu you're creating, press **Return** on the keyboard. You should notice that the first item in the list box gets selected (it will be blank at this point). The Text edit field can be used to enter the title of the menu item. You can also select the separator line to make this menu item a separator. Use separators to group related menu items and separate them from other groups. For each menu item, you can specify the command key equivalent, whether or not it has a submenu (if you're

building hierarchical menus), and its appearance (color, text style, mark symbol). Press **Return** to add another menu item. Make sure not to press **Return** after the last menu item or you'll get a blank menu item at the end of the list. To delete a menu item, simply select it and press **Delete** or choose **Clear Item** from the Edit menu. To allow you to see a preview of the menu being created, ResEdit adds your new menu to its menu bar.

There is one very important item you need to specify for each menu in your game, one that's not readily apparent from the 'MENU' dialog box. In your program code, you need a way to determine which menu was selected by the user. When the user selects an item from one of your menus, your program will be given the menu ID of the menu selected. This menu ID does not necessarily match the resource ID specified in ResEdit. To see what ID your program will receive, you need to choose **Edit Menu** and **MDEF ID** from the MENU menu. This dialog box lets you specify the ID your program will receive when an item in that menu is selected by the user. Typically it makes sense to have the menu ID match the resource ID. If you change the resource ID of a menu after it has been created, this menu ID will not change along with it. You will manually need to change it to match the new resource ID. Leave that MDEF field in this dialog box alone unless you have a 'MDEF' resource in your program you'd like to use. If you don't know what they are, don't worry. They're like 'CDEF' resources, which are described later in this chapter.

The Apple Menu

Every program needs an Apple menu as its first menu. So, you'll need to create one using ResEdit. The Apple menu, of course, contains items from the Apple menu folder. You won't need to do anything in ResEdit to define them, it's done in the code. However, you should make the last menu item entry a separator bar in preparation for adding the Apple menu folder items. Usually the only items you put in the Apple menu are the **About game...** entry and maybe a **Help...** entry.

The Ellipse ...

Any menu item command that will bring up a dialog box or prompt the user for additional information should end with three periods (an ellipse).

For example, **Save As...** should have an ellipse because it will ask the user to enter a file name, but **Save** shouldn't because it won't (with the possible exception of the first time a game is saved)

Creating an 'MBAR' Resource

An 'MBAR' resource is simply a description of your game's main menu bar. It contains a list of the 'MENU' resource IDs that correspond to the menus that will appear on the menu bar. Creating an 'MBAR' resource isn't very exciting, but it's easy. When a new 'MBAR' resource dialog box appears, it's fairly empty. All you see is that there are no menus defined and a line of asterisks next to the number 1 (see Figure 3.3).

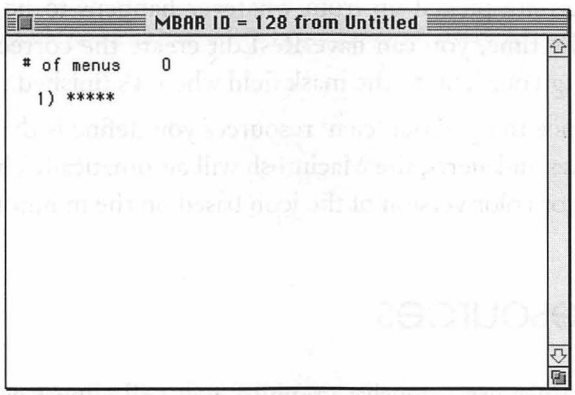


Figure 3.3 The Build Menu Var dialog box from ResEdit.

In order to specify the menus in the menu bar, you need to add fields to this resource. Click on the **asterisks** and you'll see a box drawn around them. You can now go to the Resource menu and select **Insert New Field(s)** to add an item to this list. Notice that an entry field appears titled Menu res ID. This is where you enter the resource ID of the first menu. Add as many fields as there are menus for your menu bar. You can insert new fields anywhere in your list by selecting the asterisks just above where you want the new field to appear. See, it's not too bad once you know the trick.

'ICON' and 'cicn' Resources

Icons ('ICON') and color icons ('cicn') can be defined in ResEdit and used in your game. They can be placed directly into dialog boxes or loaded and used by your program. Do not confuse these resources with the icons the Finder uses to display your game and its files. Those will be discussed in the next section.

You draw new icons using ResEdit's icon editor, a scaled-down version of MacPaint. You will notice that each icon has a mask associated with it. The icon mask is used along with the icon itself when drawing the icon. More details about masks will be given in Chapter 7, but basically, only the pixels specified in the mask get drawn from the icon itself. The other pixels are picked up from whatever happens to be on the screen. Most of the time, you can have ResEdit create the correct mask for you by dragging your icon to the mask field when it's finished.

One nice thing about 'cicn' resources you define is that, when used in dialog boxes and alerts, the Macintosh will automatically choose the black-and-white or color version of the icon based on the monitor being used.

'PICT' Resources

Because games use extensive graphics, you will almost certainly need to use picture resources. Chapter 7 explains all the details on using 'PICT' resources. To get a graphic into a 'PICT' resource, simply copy the graphic from your favorite graphics editor, and paste it into the new 'PICT' resource. The 'PICT' resource will size itself accordingly. 'PICT' resources can be used directly in dialog boxes (see the section on creating dialog boxes further on).

Finder Icons for Your Game

The Finder displays an icon for each program installed on a Macintosh. In addition, files created with that program are typically displayed with an

icon that visually associates it with the program that created it. No, this isn't Macintosh 101, but clearly the programmer must do something to tell the Finder what icons to use when displaying the program and associated files. Though not particularly difficult, you must be careful to define everything the Finder needs, otherwise you'll quickly get frustrated every time the Finder displays a generic application or document icon for your game and its files.

File Types and Creators

Every Macintosh file has a file type and creator, each a four-character field. The file creator tells the finder which application program created a particular file. The file type is used to distinguish the types of files created by an application, such as a high scores file versus a saved game file. The Finder uses both the file type and creator to determine which icon to display for a particular file. If you have the proper resources defined in your game, the Finder will display any icon you have defined to be associated with a file created by or used for your game. Also, the Finder uses this information to determine which application to launch when the user double-clicks or opens a document file.

Defining the File Type and Creator for Your Application

The development system that you're using should allow you to specify the file type and creator for your application. Typically, it can be found in the same dialog box that specifies that you are writing an application program, versus a device driver or code resource. For Think C/Symantec C++, this is defined in the dialog box displayed by selecting **Set Project Type...** from the Project menu. All application programs have a type of APPL. You get to define the creator yourself, but make sure to choose a four-character creator type that will be unique; otherwise the Finder will get confused. You will use this creator type for all files associated with your game whether the file was created manually or within the game itself.

You can also use ResEdit to specify a file's type and creator. If the resource fork is already open within ResEdit, choose **Get Info For "open file name"...** from the File menu. To change a file's type and cre-

ator without first opening it with ResEdit, select **Get File/Folder Info...** from the File menu. You will be prompted to specify the name of the file to change. Once the File Information dialog box is displayed, you can type in the file's type and creator. These values are case-sensitive.

Resource Types Needed for Finder Icons

You need to create icons for each file type your game uses, including one for the game itself. For example, Desert Trek has two file types: a high scores file, and a saved game file. This means that Desert Trek needs three Finder icons associated with it.


NOTE

Desert Trek actually creates files of a third type: the journal file and information files for help. However, these files are saved as text, and thus given a file type of TEXT. In addition, since I want TeachText (or SimpleText in more recent versions of the system) to be launched if the user double-clicks or opens one of these files, the file is given a creator type of ttxt.

To create the Finder icons and associated information for your game, you need to create resources of type 'BNDL', 'FREF', 'ICN#', 'icl4', 'icl8', 'ics#', 'ics4', and 'ics8'. Ironically, you do not use the 'ICON' resource type. That's used to define icons that your game itself can load and use. The following is a very brief description of what each resource type does:

'BNDL' The BNDL resource is what you need to create first because it will create all the other resource types for you. You specify the creator type once, called the signature by ResEdit in the dialog box displayed for BNDL resources, and as many file types as needed for your game.

'FREF' With later versions of ResEdit, you no longer need to create resources of this type yourself, ResEdit does it automatically. One resource of this type is created for each file type you define, including one for your game.

'ICN#'	This is the black-and-white version of an icon for your game or one of its associated files.
'icl4'	This is the 4-bit, or 16-color version of a Finder icon.
'icl8'	This is the 8-bit, or 256-color version of a Finder icon.
'ics#', 'ics4', 'ics8'	These are the small icons, used in the Apple and Application menus when displaying your game there.

ResEdit will also create one additional resource when you create a 'BNDL' resource for your game. This resource's type will be the same as the file signature specified in the 'BNDL' resource. This resource is known as the owner resource and is used by the Finder to display information in the **Get Info** box. Its format is described later.

Creating the Finder Icons for your Game

To create the Finder icon information for your game (you thought I'd never actually get around to telling you how, didn't you?), create a new resource of type 'BNDL'. ResEdit will display the dialog box shown in Figure 3.4.

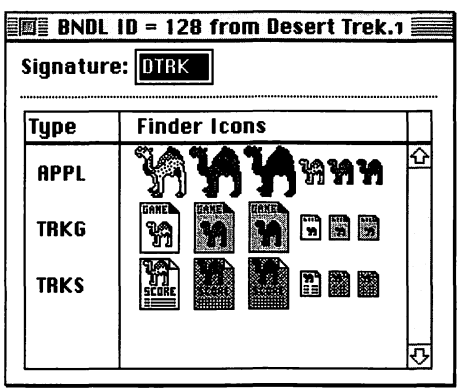


Figure 3.4 The BNDL dialog box from ResEdit.

You'll see a text edit field for the signature, and a listbox for the file types and icons. Again, the signature is a 4-character field (numerics are

allowed, too) that lets the Finder know what program created a particular file. This signature should match the file creator given to your game and should be used as the creator for all files associated with your game.

To create a new file type, select **Create New File Type** from the Resource menu. The first file type should always be APPL. The APPL file type defines the icon the Finder uses for your game. Double-clicking on the **icon boxes** under the Finder Icons heading allows you to specify or create the Finder icons for that file type. You will be prompted to specify the icon to use. Select an icon already defined, or click on the **New** button to create a new Finder icon. Remember that each Finder icon is really six icons: the black-and-white icon, the 16-color icon, the 256-color icon, and the miniversions of each. The screen that appears when you edit or create the Finder icons is sort of a scaled-down version of MacPaint. You can draw your icons here, or paste them in from a different source. Don't forget to specify the icon mask, otherwise you won't be able to see your icon. Masks are described in detail in Chapter 7, but for the moment you can simply drag one of your icons to the mask field and ResEdit will fill it in for you. For most cases, this is the mask you'll want to use anyway.

Owner Resource

ResEdit automatically creates the same type of owner's resource you type in as the game's signature in the 'BNDL' resource. The owner resource is used by the Finder to display Get Info information. This resource is simply a text string describing your game. Opening this resource shows you the generic ResEdit resource editor. The leftmost column shows the data offset starting at zero, the middle column shows hexadecimal data, and the rightmost column shows ASCII data. You can type hexadecimal numbers in the middle column, text in the right column.

The owner resource is a Pascal string, meaning the first byte is a length indicator. Starting at the second byte, or character, of the right column, type in the descriptive text for your game. After typing it in, count how many characters are in the description, convert that to hex,

and type in that number as the first byte in the hexadecimal column. If you don't feel like using the data offset column and counting to figure out the length of your text, you can close the resource, look at its size and subtract one (for the length byte itself).

Creating Dialog Boxes for Your Game

One of the more time-consuming tasks of writing a game is designing and building the dialog boxes. Some games require much more work in this area than others, but nearly every game is going to need multiple dialog boxes. For example, Desert Trek has fifteen dialog boxes. The creation of dialog boxes using ResEdit will be discussed here. An explanation of the different types of dialog boxes along with rules for their proper use can be found in Chapter 6.

Dialog boxes use several resource types, depending on what you define. The dialog box definition itself is contained in the 'DLOG' resource. Every dialog box, of course, contains controls, or dialog box items. The definitions of these items are contained in the 'DITL' resource. If you define custom colors for the dialog box, their definitions are stored in the 'dctb' resource. You typically will not need to define the 'DITL' and 'dctb' resources yourself, because they will automatically be created when you use ResEdit's dialog box editor to create a 'DLOG' resource. You can also create custom colors for dialog box items by creating an 'ictb' resource for each dialog box. The 'ictb' resource is somewhat cryptic, and ResEdit does not provide a way to graphically edit dialog box item colors. However, with a lot of patience, and a little trial and error, ictb resources can greatly enhance the appearance of your games.

The 'DLOG' Resource

The 'DLOG' resource is where you can define most elements related to a dialog box. Figure 3.5 shows the dialog box editor ResEdit displays when you create or edit a dialog box.

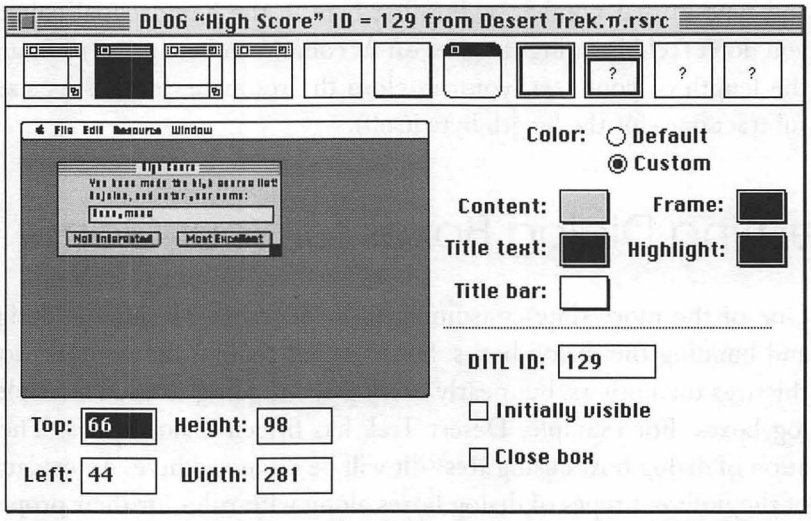


Figure 3.5 The DLOG dialog box editor from ResEdit.

From here, you can specify the window used to display the dialog box, including whether or not it has a close box, scroll bars, or even a title bar, and its size and location. To specify the dialog box's title in the title bar, you need to select **Set 'DLOG' characteristics** from the DLOG menu. Don't forget to do this; otherwise your dialog box will have no title, or, if you're like me, and you copy and paste dialog boxes from one place to another, have the completely wrong title.

Typically, when my games display a dialog box, they get centered either on the screen or on my main game window (which the user can move around). For this reason, specifying the dialog box's location isn't necessary. Also, the dialog box's visible flag should be off, so that the dialog box isn't first displayed at the location specified in ResEdit, then moved to the center of the screen. Remember, the game player's screen may not be the same size as your screen, so you may always want to have your program position the location of dialog boxes. If your game requires system 7.0 or later, you can use ResEdit to automatically have the dialog box centered on the main screen, regardless of its size. Select **Auto Position...** from the DLOG menu to do this.

One of the more important things you need to select here is the DITL ID. ResEdit defaults to a DITL ID equal to that of the dialog box ID. For most cases, that's just fine. However, you need to be aware that it isn't always the case. For example, if you copy and paste a dialog box from one program to another, something I frequently do to save time when writing a new game that can use a dialog box similar to one I've defined for another game, just copying the 'DLOG' resource isn't enough. You will need to copy the 'DITL' resource associated with that dialog box, as well as any other supporting resources such as the 'dctb' and 'ictb' resources. When doing so, make sure all the IDs match up.

Another potential snag relates to 'ALRT' resources. Alerts, of course, are special dialog boxes frequently used by many programs. For reasons I'll explain later, I prefer not to use alerts, but if you decide to use them, keep in mind that they also use 'DITL' resources to describe their items. You will need to be careful because ResEdit will start assigning IDs of 128 to both 'ALRT's and 'DLOG's, and the 'DITL's they use. Obviously, the 'ALRT' with ID 128 and the 'DLOG' with ID 128 can't use the 'DITL' resource of ID 128. You will need to rectify the situation by renumbering either the 'ALRT's or 'DLOG's. I strongly recommend that a 'DITL's resource ID matches that of its owning 'ALRT' or 'DLOG'. Otherwise, you are going to be spending a lot of time sometime in the future trying to figure out why you can't get that dialog box or alert to work properly.

When you delete a 'DLOG' or 'ALRT' from your game, be careful to delete all the related resources such as the 'DITL', 'ictb', and 'dctb' ('ALRT's use the 'actb' for custom color definitions). Otherwise, you will have orphan resources lying around in your game, eating up disk space and waiting to cause potential problems in the future.

The 'DITL' Resource

As previously explained, the 'DITL' resource contains information describing all the items contained in a dialog box or alert. When creating a dialog box, you will probably spend most of your time creating and positioning the dialog box items. To edit the 'DITL', simply double-click

on the miniature representation of the dialog box you're building in the left side of the Dialog dialog box (try not to get confused, but remember, you are using a dialog box to build a dialog box). Doing so will display a full-sized representation of the dialog box you're building, along with a floating control palette.

To add controls to your dialog box, drag and drop controls from the palette to your dialog box. Double-click on a control to set its properties, such as text. You can move a control by dragging it around, and you can size it by selecting it and using the minisize control (a small black box in the lower-right corner). The alignment menu allows you to align several controls with relation to each other.

To add an icon or picture to your dialog box, you need to first define the 'PICT' or 'ICON' resource (color icons have a resource type 'cicn') to be used because you'll be asked to enter its ID in ResEdit's Properties dialog box. For icons, use the same ID for the black-and-white 'ICON' resource and color 'cicn' resource. This way, the correct icon will automatically be displayed when the dialog box is shown on black-and-white or color monitors.

The item numbers are very important. First, they are the IDs that your code will use to manipulate the item, such as setting and reading text from an edit field. They also determine the tabbing order. The first edit field to contain the cursor will be the one with the lowest item ID. When the user presses the **Tab** key, the edit fields will be traversed in order according to their IDs.

For more information about the types of controls that can be used, see Chapter 6. There are a few special tricks that can be used to create better-looking dialog boxes, such as custom control colors and the use of control definitions, or 'CDEF's, to give buttons a three-dimensional look. Since no coding is necessary to achieve these results, they will be described here.

3D Buttons Using a 'CDEF'

Control definitions are functions that are used to draw a control. The Macintosh operating system calls the CDEF function whenever a control

needs to be drawn on the screen. Apple provides a default 'CDEF' to draw all controls. It has an ID of 0, and all controls you create have a default ProcID of 0. The ProcID field of a control tells the Macintosh what 'CDEF' to use when drawing that control. By changing the **ProcID** of a control used in your dialog box, you can specify a different 'CDEF' for drawing your controls.

Desert Trek uses a 'CDEF' written by another author who has been kind enough to allow me to use it royalty-free. The 'CDEF' draws push buttons, radio buttons, and check boxes with a three-dimensional effect. The 'CDEF' can be found on the CD included with this book. See Appendix B for more details.

There are two basic approaches that can be used to include the 3D buttons 'CDEF' into your program. The easy way is to copy the 'CDEF' into your game's resource file and give it an ID of 0. Because the standard Macintosh 'CDEF' has an ID of 0, the one included in your game will automatically override it (only for your game, not the entire system). This means that any buttons (push, radio, or check box) displayed by your program will be drawn with the 3D effect. Pretty easy, huh? However, there's a catch: any button drawn will have the 3D effect, even those in System dialog boxes called by your program such as the standard Macintosh File Open and Save dialog boxes. Needless to say, this is not the Macintosh way. You can customize the interface to your program, within reason, but you should never affect standard system operations or appearance without the user's express written consent. With a little bit of work, you can easily overcome this problem by implementing 3D controls using the method described as follows.

'CNTL' Resources

'CNTL' resources are used to define generic controls that can be used in your game's windows and dialog boxes. ResEdit allows you to place these controls in dialog boxes. However, before doing so, you must obviously define the control first. Creating a new 'CNTL' resource produces the dialog box seen in Figure 3.6.

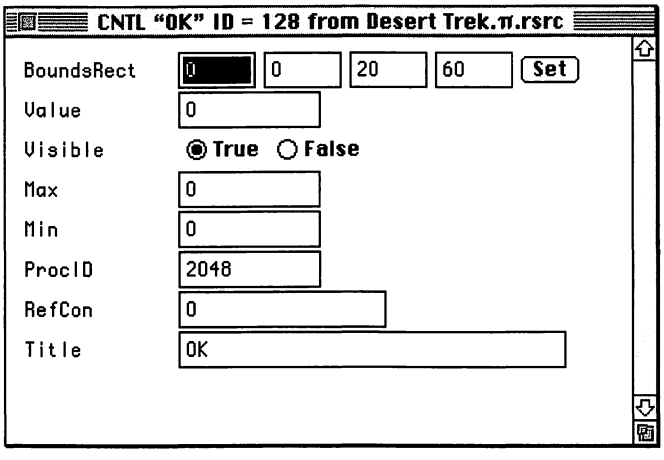


Figure 3.6 The CNTL editor from ResEdit.

Of the many options that you can set here, only a few are of interest for the purpose of setting up the 3D control. First, you need to specify the dimensions of the control. The **BoundsRect** property is used to specify the size and position of the control. However, if you think about it, you can't always tell where in a dialog box you'll want the control to go. Furthermore, what if you want to use this control (such as an **OK** button) in multiple dialog boxes, but at different positions? Fortunately, ResEdit allows you to place the control into a dialog box at any position, regardless of what the **BoundsRect** property states. Thus, for controls you plan on using ResEdit to place into dialog boxes, it makes the most sense to use the **BoundsRect** property to specify only the control's width and height. This means that the top and left values should be 0, and that the bottom and right values should specify the control's height and width (the four **BoundsRect** values are entered in this order: top, left, bottom, right). Unfortunately, ResEdit does not allow you to change the control's width and height when placing it into a dialog box, so if you want the same control to be a different size in two different dialog boxes, you'll need to create two different controls (for example, if you want a large **OK** button in one dialog box and a small one in another).

You need only specify two more fields to create your 3D control. The easy one is the **Title** property, which specifies the text displayed for the con-

trol. This is the text that appears in a push button, or next to a radio button and check box control. The less obvious one is the **ProcID** property.

The ProcID Property

The **ProcID** property specifies which 'CDEF' to use when drawing the control. The 'CDEF' specified will determine the type of control you get: button, scroll bar, etc. The standard Macintosh push button has a **ProcID** of 0, the radio button an ID of 2, the check box an ID of 1, and the scrollbar an ID of 16. To use a 'CDEF' that you have copied into your game's resource file, you need to specify its ID here. However, there's a small catch. The **ProcID** isn't exactly the resource ID of the CDEF. For starters, remember that the push button, radio button, and check box all use different **ProcID**s (0, 2, and 1, respectively). However, the single 3D 'CDEF' used in Desert Trek draws all three types of buttons. How does one CDEF cover all three types of buttons, and more importantly, how do you, the game programmer, specify which one to use? The answer is apparent in Figure 3.7.

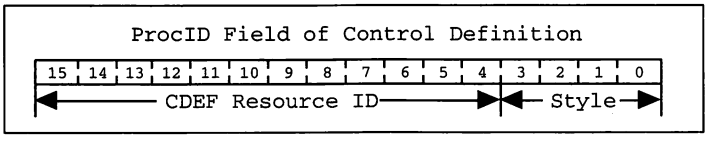


Figure 3.7 The **ProcID** bits specification.

The 'CDEF' resource ID is only a portion of the **ProcID**. To understand this a little better, you need to know that certain controls are grouped together. For example, push buttons, radio buttons, and check box controls are considered to be related. They are all styles of buttons. The Macintosh uses only one 'CDEF' to draw all controls in a group. The CDEF knows exactly what button to draw by looking at the control style passed to it by the Macintosh when it's time to draw a control. How does the Macintosh know what style to send the 'CDEF'? It uses the last 4 bits of the **ProcID** property. So now you know that standard scroll bars, which use a **ProcID** of 16, are considered a different type of control than buttons, and the default Macintosh 'CDEF' for drawing them is... (drum roll, please) 1 (see Figure 3.8).

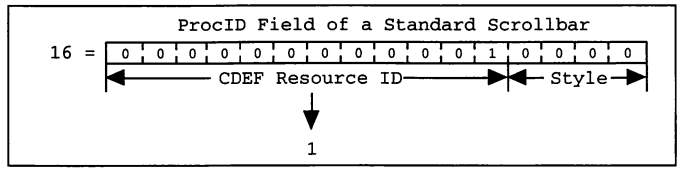


Figure 3.8 The **ProcID** of 16 for scroll bars equates to a 'CDEF' ID of 1.

To determine what **ProcID** to use when you want to specify a 'CDEF' like the 3D CDEF Desert Trek uses, you need to multiply the CDEF resource ID by 16. Then, add 1 for a check box, or 2 for a radio button. Thus, because the 3D buttons 'CDEF' Desert Trek uses has a resource ID of 128, a push button would need a **ProcID** of 2048 (128 * 16), a radio button 2050 (128 * 16 + 2), and a check box 2049 (128 * 16 + 1).

Using 'CNTL' Resources in Your Dialog Boxes

It is very simple to use 'CNTL' resources in your dialog box. When editing the dialog box items (the 'DITL' resource), use the control palette to place a "Control" control on your dialog box. Open its properties and you'll see a dialog box displayed. The only thing you really need to specify here is the 'CNTL' resource ID of the control you wish to use. You can move the control anywhere in the dialog box you're creating, just like any of the other standard controls; however, you will not be able to change its size. The size would need to be changed back in the original 'CNTL' definition.

Final Notes on Using the 3D Button 'CDEF'

For best visual results, you will want to give most dialog boxes using the 3D button 'CDEF' a custom background color of light gray. You specify this color in the 'DLOG' resource. See Desert Trek's resource file for the exact shade of light gray used.

Lastly, if multiple dialog boxes use the same button, such as an **OK** button, you need to define only one 'CNTL' resource. You can then use that resource in as many dialog boxes as you want. The only restriction is

that the button must be the same size in all dialog boxes because the size can only be specified in the 'CNTL' resource itself.

Custom Colors and Font Styles for Dialog Box Items

An easy way to spice up the appearance of your application is to add color to the controls found within your dialog boxes. The push buttons, radio buttons, and check boxes already look nice thanks to the 3D button 'CDEF', but what about the static text and text edit controls? Without any coding, you can add custom colors and change font styles for these controls by adding an 'ictb' resource for each dialog box. In order to use an 'ictb' resource to define colors and text styles for dialog box items, you must first define a custom color resource for your dialog box (the 'dctb').

The 'ictb' resource defines custom styles for each control in the dialog box with the same resource ID as itself. There is no built-in ResEdit editor for this resource type, so you will need to type all the data by hand into ResEdit's generic resource editor described in the preceding Owner Resource section. The format of the 'ictb' resource is, to be quite honest, not trivial to build, but plenty of examples will be given after a complete explanation of its format.

The 'ictb' resource is divided into two sections. The first section contains a pair of words for each item in the dialog box. The two words specify what styles are to be changed for a dialog box item, as well as the offset within the 'ictb' where the style record for that dialog box item can be found. The second part of the 'ictb' resource is an array of style records defining the actual styles for the dialog box items. Two dialog box items can use the same style record if they are to have the same style.

Style Definition and Offset Section

Each item in the dialog box must have an entry in this section of the 'ictb' resource, even if you are not changing the style of that item. Each entry is 4 bytes, or two words. The first entry, bytes 1 through 4, pertain to dialog box item 1, the second entry, bytes 5 through 8, pertain to dialog box item 2, and so on. Thus, if a dialog box has six items, this section will be 24 bytes in length.

The first word of an entry, the item data word, contains a specification of the styles that are to be changed for the dialog box item. The text's font, size, style, and color can be changed, in any combination. The following table shows what values for the item data word affect what characteristics of the item:

Characteristic	Word Value (Hexadecimal)
Text Font	0001
Text Style	0002
Text Size	0004
Foreground Color	0008
Add to Text Size	0010
Background Color	2000
Text Mode	4000
Font is Name	8000
Text Font	You can change the text font by using this value. The font can be specified in one of two ways in the style record (which will be discussed shortly). You can either specify a font number, or a font name. Though slightly more complex, it is strongly recommended that you use the font name method because you cannot always rely on font numbers to be the same from system to system.
Text Style	The text style includes the following characteristics: bold, italic, underline, outline, shadow, condensed, and extended.
Text Size	This will affect the size of the text.
Foreground Color	The will change the color of the text itself. You will specify the RGB value in the style record.
Add to Text Size	Rarely used, this characteristic allows you to add to the standard text size instead of specifying it directly.

Background Color	This will change the color of the background behind the text.
Text Mode	This affects the method by which the text is drawn. See Chapter 7 for a complete explanation of bit transfer modes, such as srcOr, srcXor, and srcCopy.
Font is Name	This value, when set, specifies that the font name will be used instead of the font number. See the following section on style records for more information. Again, if you plan to change the font, this method of font name specification is strongly recommended over the font number method since font numbers may not always be the same from one Macintosh to the next.

To change more than one style, simply add together the values. For example, to change the text style and foreground color, add 0x0002 and 0x0008 to get 0x000A. To change the text font and size, add 0x0001, 0x0004, and 0x8000 to get 0x8005.

The second word of an entry, the item data offset, specifies where, in the 'ictb' resource, the style record begins. The first style record will start immediately following the item data and offset section. So, for a dialog box with six items, the first style record will begin at byte 24 (0x18). Since each style record is 20 bytes in length, the second style record will begin at 44 (0x2C), the third at 64 (0x40), etc. Again, only those items you wish to change the style of need style records, so there will almost always be fewer style records than items in your dialog box. Also, remember that two dialog box items can share the same style record, meaning that the item data offset word can be the same for multiple dialog box items.

The only dialog box item types that we plan to change with the 'ictb' resource are static text and text edit fields. So, for every other type of dialog box item such as buttons, icons, and pictures, the entry for that control will be 0000 0000 (4 bytes of zeros).

Style Records

The second section of the 'ictb' resource contains all the style records. Each dialog box item style record is 20 bytes in length. The item data word determines which fields will be used in the style record. So, even if you define a foreground color in the style record, it will only be used if the appropriate value is set in the item data word.

The following table describes the format of a 20-byte text style record:

Offset	Size	Description
0	2	Text Font (Font Number, or Offset to Font Name)
2	2	Text Style
4	2	Text Size
6	6	Foreground Color (RGB, 2 bytes for each component)
12	6	Background Color (RGB, 2 bytes for each component)
18	2	Font Mode

Text Font

This can be the text font number, or an offset to the text font name. The first text font name immediately follows the last style record. So, for a dialog box with six items and two style records, the first font name offset would be (6 * 4) + (2 * 20), or 64 (0x40). The font name specified is a Pascal string, which means the first byte of the string is a length indicator. So, if you wanted to specify the font name Geneva, the text font name would start with 0x06, followed by the ASCII string Geneva (which can be typed directly in the rightmost field of RedEdit's generic resource editor).

Text Style

This word specifies the style of the text, such as bold and italics. the following table shows the values you need to use:

Style	Word Value (Hexadecimal)
Bold	0100
Italic	0200
Underline	0400
Outline	0800
Shadow	1000
Condensed	2000
Extended	4000

Add the values together to combine styles. For example, bold and underlined would be specified by using a value of 0x0500 (0x0100 + 0x0400).

Text Size

This field specifies the size of the text. If the add to text size value is set in the item data word, this value will be added to the current text size.

Foreground Color

This 6-byte field specifies the color of the text. The red component is specified in the first 2 bytes, green in the second 2, and blue in the last 2 bytes. For example, a dark blue would be specified as 0000 0000 8000 (values are in hexadecimal).

Background Color

This 6-byte field specifies the background color to be used behind the text.

Font mode

This value specifies the transfer mode to be used when drawing the text. You should never need to use this field.

Example 1: Desert Trek's Bad Name Dialog Box (ID=133)

The Bad Name dialog box is displayed if, when the user is typing in a name for the high scores list, the typed-in name has fewer than 1 or more than 20 characters. There are only three dialog box items: a 3D push button, a static text field, and an icon. The only style change I want to make is to have the static text color appear dark blue.

Remember, the first section of the 'ictb' resource contains two words for each dialog box item: the item data word and the item data offset. Since there are three dialog box items, this section will be 12 bytes in length. This also means that the first, and in this case, only style record will start at offset 12 (0x0C).

Since the only item's style I want to change is the static text field (item ID 2), the other items will have entries of 0000 0000 in this section. In addition, the only style I want to change for the static text field is the foreground color, so the item data word needs to be 0x0008. Therefore, the first section of the ictb should look like this (the first word is the item data word, the second word is the item data offset):

Dialog Box Item	ictb Offset	Entry (Hexadecimal)
1 (Push Button)	0 (0x00)	0000 0000
2 (Text Edit Field)	4 (0x04)	0008 000C (change foreground color)
3 (Icon)	8 (0x08)	0000 0000

The single style record for this ictb, which begins at offset 0x0C, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	12 (0x0C)	0000
Text Style	14 (0x0E)	0000
Text Size	16 (0x10)	0000
Foreground Color	18 (0x12)	0000 0000 8000

Background Color	24 (0x18)	0000 0000 0000
Text Mode	30 (0x1E)	0000

See Figure 3.9 for the Bad Name 'DITL' resource showing the dialog box item numbers, and the complete 'ictb' resource as described previously.

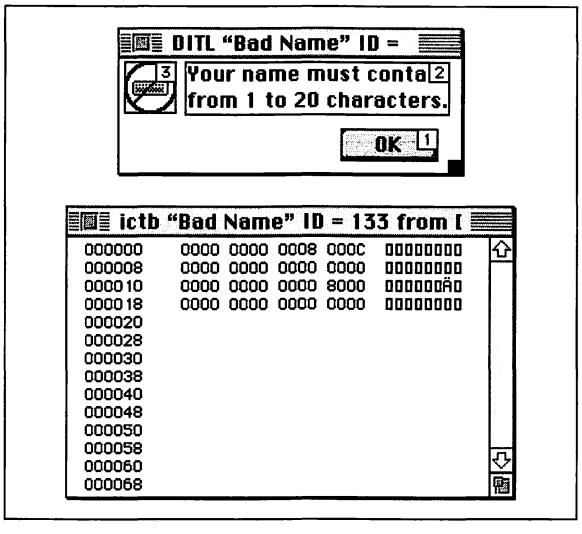


Figure 3.9 The DITL and ictb resources for the bad name dialog box.

Example 2: Desert Trek's High Score Dialog Box (ID=129)

The High Score dialog box allows the game player to enter his or her name when a new high score is achieved. The dialog box has five items: an icon, two 3D push buttons, a static text field, and a text edit field. I want to change the static text field to use a text color of dark blue, and the text edit field to use a background color of yellow. The item ID of the static text field is 4, and the ID of the text edit field is 3.

Since there are five dialog box items, the first section of the 'ictb' resource will be 20 bytes. This means that the first style record will start at offset 20 (0x14). Now, I'm going to use a neat trick to reduce the size of the 'ictb'. You would initially think that I need two style records, one for the static text field and one for the text edit field. However, since I'm

changing only the foreground color for the static text field, and only the background color for the text edit field, I can use the same style record for both. The style record itself would contain data for both the foreground and background colors, but the item data word for the static text will only pick up the foreground color, and the item data word for the text edit field will only pick up the background color.

The first section of the 'ictb' would look like this:

Dialog Item	ictb Offset	Entry (Hexadecimal)
1 (Push Button)	0 (0x00)	0000 0000
2 (Push Button)	4 (0x04)	0000 0000
3 (Text Edit Field)	8 (0x08)	2000 0014 (change background color)
4 (Static Text Field)	12 (0x0C)	0008 0014 (change foreground color)
5 (Icon)	16 (0x10)	0000 0000

The single style record for this 'ictb', which begins at offset 0x14, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	20 (0x14)	0000
Text Style	22 (0x16)	0000
Text Size	24 (0x18)	0000
Foreground Color	26 (0x1A)	0000 0000 8000
Background Color	32 (0x20)	FFFF FFFF 8000
Text Mode	38 (0x26)	0000

See Figure 3.10 for the 'DITL' resource showing the dialog box item numbers, and the complete ictb resource as described previously.

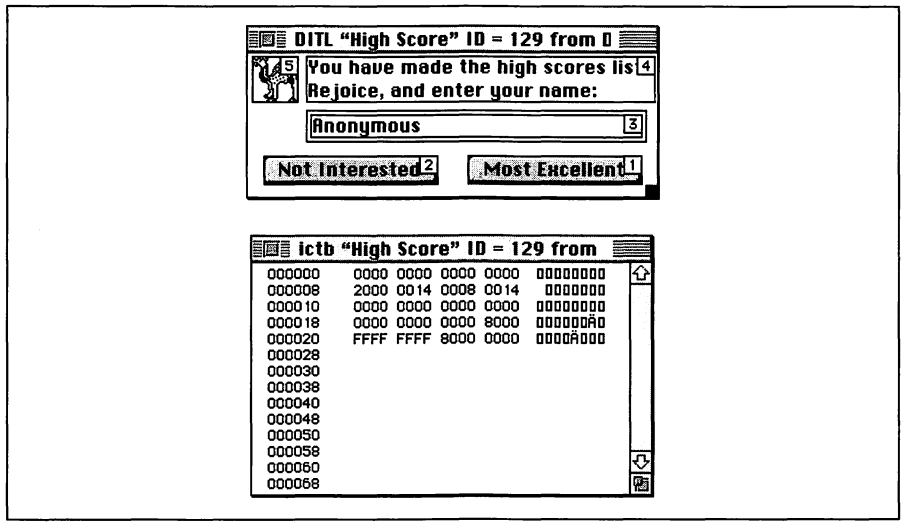


Figure 3.10 The DITL and ictb resources for the High Score dialog box.

Example 3: Desert Trek's About... Dialog Box (ID=135)

The About Desert Trek... dialog box is shown when the user chooses **About Desert Trek...** from the Apple menu. This dialog box has nine items: two 3D push buttons, six static text fields, and one icon. Now I want to start getting fancy, and use different fonts, styles, and colors for the static text fields. Here's what I want to do to the following static text fields:

ID	Styles Wanted
4	Text color blue. Times font.
5	Text color red. Geneva font. Size 10.
6	Text color red. Geneva font. Size 10. Bold.
7	Text color blue. Geneva font. Size 10. Bold.
8	Text color blue.
9	Text color blue.

The first question is, How many style records are needed?. If I wanted to be safe, I could create one style record for each static text field. However, it looks like I can consolidate several fields very easily. First, static text fields 8 and 9 have exactly the same style, so they can be combined into one style record. Furthermore, static text item 7 has the same color as items 8 and 9. Since items 8 and 9 only define the foreground color, the other fields of their style record will be ignored. Thus, I can define a style record for item 7, and have items 8 and 9 pick up only the foreground color field from that style record. Using the same reasoning, it looks as if I could also combine the style records of static text fields 5 and 6. The only difference between the two is the bold text style, which won't be specified in item 5's item data word. However, there's a catch here: both items 5 and 6 specify a font. It seems that the dialog box would get confused if two items with slightly different styles but the same font use the same style record. In this case, if items 5 and 6 used the same style record, the font attribute for item 6 would not be picked up correctly. It's one of those unfortunate facts of life. Since items 7, 8, and 9 will be sharing a style record, only four style records will be needed.

There are nine items in the dialog box, so the first style record will begin at 36 (0x24), the second at 56 (0x38), the third at 76 (0x4C), and the last at 96 (0x60). Notice that I'll also need to define two fonts by name, Times and Geneva. The first font name will appear immediately after the last style record, which will be at offset 116 (0x74). The first font, Times, has five characters. This means that the font entry will be six characters in length (remember, the font entry is a Pascal string, which means that the first byte is a length indicator). The second font will thus begin at 116 + 6, or 122 (0x7A).

The first section of the 'ictb' would look like this:

Dialog Box Item	ictb Offset	Entry (Hexadecimal)
1 (Push Button)	0 (0x00)	0000 0000
2 (Push Button)	4 (0x04)	0000 0000
3 (Icon)	8 (0x08)	0000 0000

4 (Static Text Field)	12 (0x0C)	8009 0024 (foreground color, font)
5 (Static Text Field)	16 (0x10)	800D 0038 (foreground color, font, size)
6 (Static Text Field)	20 (0x14)	800F 004C (foreground color, font, size, style)
7 (Static Text Field)	24 (0x18)	800F 0060 (foreground color, font, size, style)
8 (Static Text Field)	28 (0x1C)	0008 0060 (foreground color)
9 (Static Text Field)	32 (0x20)	0008 0060 (foreground color)

The first style record, starting at offset 36 (0x24) and specifying a Times font and a foreground color of blue, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	36 (0x24)	0074 (Offset to font name)
Text Style	38 (0x26)	0000
Text Size	40 (0x28)	0000
Foreground Color	42 (0x2A)	0000 0000 8000 (Blue foreground)
Background Color	48 (0x30)	0000 0000 0000
Text Mode	54 (0x36)	0000

The second style record, starting at offset 56 (0x38) and specifying a Geneva font, a text size of 10, and a foreground color of red, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	56 (0x38)	007A (Offset to font name)
Text Style	58 (0x3A)	0000
Text Size	60 (0x3C)	000A (Font size of 10)

Foreground Color	62 (0x3E)	8000 0000 0000 (Red foreground)
Background Color	68 (0x44)	0000 0000 0000
Text Mode	74 (0x4A)	0000

The third style record, starting at offset 76 (0x4C) and specifying a Geneva font, a text size of 10, a text style of bold, and a foreground color of red, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	76 (0x4C)	007A (offset to font name)
Text Style	78 (0x4E)	0100 (bold)
Text Size	80 (0x50)	000A (font size of 10)
Foreground Color	82 (0x52)	8000 0000 0000 (red foreground)
Background Color	88 (0x58)	0000 0000 0000
Text Mode	94 (0x5E)	0000

The fourth and last style record, starting at offset 96 (0x60) and specifying a Geneva font, a text size of 10, a text style of bold, and a foreground color of blue, looks like this:

Style Record Field	ictb Offset	Entry (Hexadecimal)
Text Font	96 (0x60)	007A (offset to font name)
Text Style	98 (0x62)	0100 (bold)
Text Size	100 (0x64)	000A (font size of 10)
Foreground Color	102 (0x66)	0000 0000 8000 (blue foreground)
Background Color	108 (0x6C)	0000 0000 0000
Text Mode	114 (0x72)	0000

The first font text description, Times, begins at offset 116 (0x74), and looks like this:

ictb Offset	Entry (Hexadecimal)
116 (0x74)	0554 696D 6573 (length of 5, ASCII values for Times)

The second font text description, Geneva, begins at offset 122 (0x7A), and looks like this:

ictb Offset	Entry (Hexadecimal)
122 (0x7A)	0647 656E 6576 61 (length of 6, ASCII values for Geneva)

See Figure 3.11 for the 'DITL' resource showing the dialog box item numbers and the complete ictb resource as described previously.

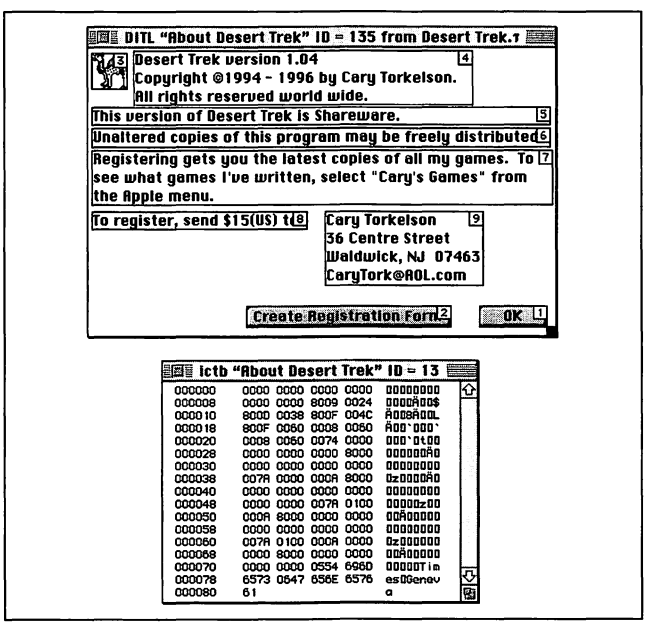


Figure 3.11 The DITL and ictb resources for the About Desert Trek... dialog box.

'WIND' Resources

Window resources define windows that your program can load. Think of windows as dialog boxes without any predefined controls. ResEdit allows you to create windows in a very similar manner to dialog boxes, except that you don't define a 'DITL'. Desert Trek uses windows for its main screen and high scores screen.

'TEXT' and 'styl' Resources for Styled Text

The 'TEXT' and 'styl' resources can be used to display bulk text messages to the user, such as on-line help. The nice thing about these resources is that you can display formatted text with multiple fonts and styles easily in a TextEdit window (see Chapter 8 for more details on how to load and use these resources). You should not edit the 'styl' resource directly because ResEdit automatically creates one when you edit a 'TEXT' resource. However, keep in mind that if you copy or renumber a 'TEXT' resource, you need to do the same with the corresponding 'styl' resource (they will have the same resource ID).

When you create a 'TEXT' resource, ResEdit provides a very simple text editor that allows you to type in the text as well as change the font, size, and style. The 'TEXT' resource has a limit of about 32,000 characters, so you'll need multiple 'TEXT' resources for text blocks greater than that.

The 'SIZE' Resource

The Finder uses the 'SIZE' resource to determine how much memory to give an application when it runs. There are two size parameters specified in this resource: the preferred memory size and the minimum memory

size. In addition to these two memory specifications, there are 16 bits of information pertaining to your game, such as whether or not your game is 32-bit clean.

You will need to determine how much memory your application requires to run. This will most certainly be affected by many factors, such as the number of offscreen bitmaps your game uses. Several methods can be used to determine your game's memory usage, and it's up to you to find out the minimum needed to run your game. For example, you could give your game as much memory as your Macintosh allows, and use the **About This Macintosh** from the Apple menu of the Finder to see how much your game is actually using. Be careful, though, because this number fluctuates based on what your program has loaded (in terms of dialog boxes, windows, files, etc.), as well as factors such as the monitor's pixel depth. Try to maximize your game's memory usage before getting a reading, possibly by opening as many windows as possible.

When creating a 'SIZE' resource, you need to give it an ID of -1. If the user changes the memory settings via the Get Info window from the Finder, these settings will be stored in new 'SIZE' resources of IDs 0 and 1 (the new preferred size can be found in ID 0, the new minimum size in ID 1). The memory settings are specified in bytes, so if you want a minimum and preferred size of 1024K, which Desert Trek uses, the values specified would be 1024 * 1024, or 1048576.

The Version (vers) Resource

The 'vers' resource provides the Finder with information concerning the version of your program. If properly defined, the information will be displayed when the user chooses **Get Info** from the File menu with your game selected. Version information will also be displayed if the user chooses the **Show Version** option from the Views Control Panel.

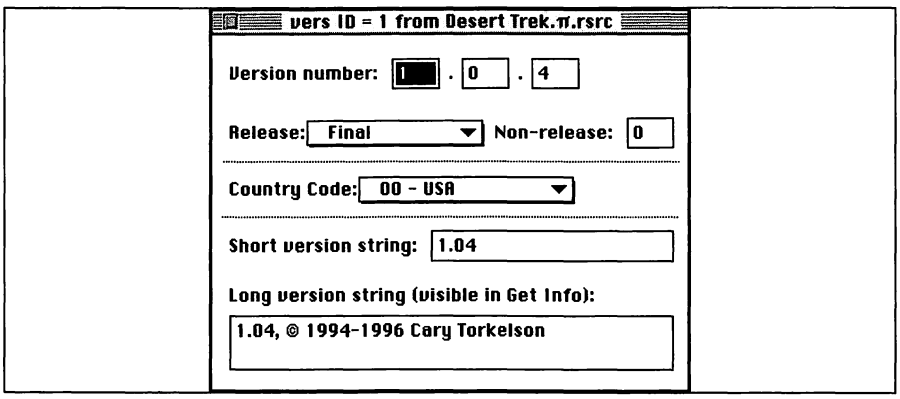


Figure 3.12 The vers resource editor from ResEdit.

ResEdit provides a simple-to-use 'vers' resource editor for defining the resource, as shown in Figure 3.12. The Finder will use version information from 'vers' resources with IDs of 1 and 2. The Finder will use the short version string from vers resource ID 1 to display the version number of your game for all Finder folder views except icon and small icon (assuming, of course, that the user has chosen to view version information from the Views Control Panel). The long version string of 'vers' resource ID 1 is used in the Get Info window and will be displayed at the Version label. Note that if you do not have a 'vers' resource of ID 1, the Finder uses the Owner Resource to display information here. Lastly, the Finder will use the long version string of vers resource ID 2, if it exists, to place additional information about your application as part of the title near the game's icon at the top of the Get Info window. See Figure 3.13 for a summary of the information that is used and where it goes.

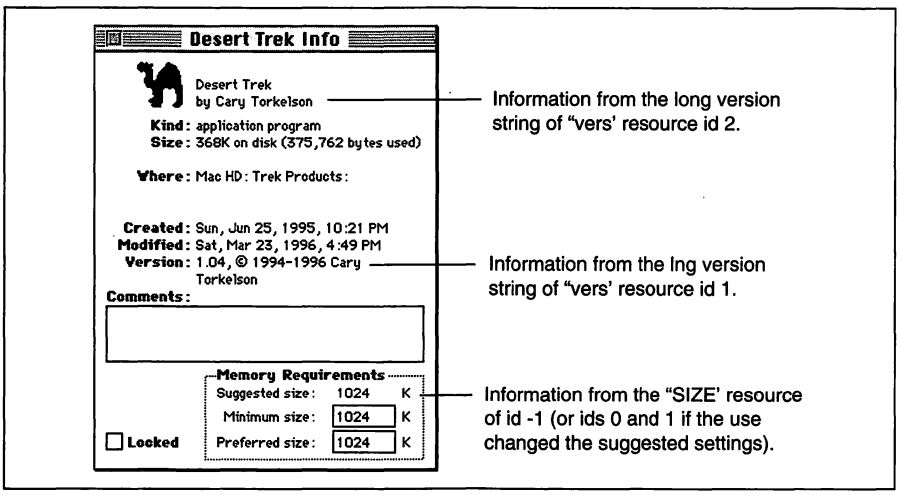


Figure 3.13 The Get Info window for Desert Trek.

'STR#' Resources

'STR#' resources are used to define the text strings your program uses. It is good practice to include all textual output strings, including even those of just one word, in the resource file. If you need to change these strings for any reason, such as supporting a new language or fixing a typo, you only need to change the resource fork of your game. If the strings were hard coded into your program, you would need to recompile in order to have the changes take affect. This is especially important when someone else needs to change the strings. They won't need your source code.

Strings are stored in groups, each group being a single 'STR#' resource. In other words, a single 'STR#' resource contains multiple strings. When you load a string from the resource fork of your game, you will need to specify the resource ID of the 'STR#' resource and the offset within that resource.

When you create a new 'STR#' resource, it will contain no strings. Like the 'MBAR' resource, you need to add strings to the list before you can type them in. Do so by clicking on the ***** next to a number, and select **Insert New Field** from the Resource menu. A blank field will be inserted, where you can type in the new string.

To read strings into your program, you use the `GetIndString()` toolbox call, which has the following parameters:

```
GetIndString(   Str255   str255,
                short    sResourceID,
                short    sStringOffset );
```

Typically, you will want to read a string once and use it many times. This is especially true for strings used to update the screen, since you will not want to load them in every time you draw the screen. In order to minimize the storage that your strings require once loaded in memory, you should use a temporary variable to load the string from the resource file, and use the `NewString()` toolbox call to duplicate the string just loaded in memory:

```
StringHandle NewString( Str255 str255 );
```

A variable of type Str255 requires 256 bytes of storage (255 characters plus 1 length byte), regardless of the length of the string itself. `NewString()` allocates just enough storage to hold the string. This can significantly reduce the storage required to hold your program strings, especially when many of them are small. However, you should not modify a string created with `NewString()` because if you attempt to make it larger, you will corrupt memory (that's a bad thing).

Desert Trek's game window uses a number of strings, loaded from a 'STR#' resource, to display game status. The code to define and load the strings looks like this:

```
#define TREK_WINDOW_STRINGS_ID        128
#define HUNGER_STRING_ID                1
#define THIRST_STRING_ID                2
#define FATIGUE_STRING_ID               3
#define HEALTH_STRING_ID                4
#define FOOD_STRING_ID                  5
#define WATER_STRING_ID                 6
#define ELIXERS_STRING_ID               7
#define CANNIBALS_DISTANCE_STRING_ID    8
#define GOLD_STRING_ID                  9
#define DISTANCE_TRAVELLED_STRING_ID   10
#define KM_STRING_ID                   11
#define BEHIND_YOU_STRING_ID           12
#define SCORE_STRING_ID                13

static StringHandle  hStringHunger;
static StringHandle  hStringThirst;
static StringHandle  hStringFatigue;
static StringHandle  hStringHealth;
static StringHandle  hStringFood;
static StringHandle  hStringWater;
static StringHandle  hStringElixers;
static StringHandle  hStringCannibalsDistance;
static StringHandle  hStringGold;
static StringHandle  hStringDistanceTravelled;
static StringHandle  hStringKm;
static StringHandle  hStringBehindYou;
static StringHandle  hStringScore;

void SetTrekWindowStrings( void )
{
  Str255  str255;

  GetIndString( str255, TREK_WINDOW_STRINGS_ID, HUNGER_STRING_ID );
  hStringHunger = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, THIRST_STRING_ID );
  hStringThirst = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, FATIGUE_STRING_ID );
  hStringFatigue = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, HEALTH_STRING_ID );
  hStringHealth = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, FOOD_STRING_ID );
  hStringFood = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, WATER_STRING_ID );
  hStringWater = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, ELIXERS_STRING_ID );
```

```
  hStringElixers = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID,
                CANNIBALS_DISTANCE_STRING_ID );
  hStringCannibalsDistance = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, GOLD_STRING_ID );
  hStringGold = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID,
                DISTANCE_TRAVELLED_STRING_ID );
  hStringDistanceTravelled = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, KM_STRING_ID );
  hStringKm = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, BEHIND_YOU_STRING_ID );
  hStringBehindYou = NewString( str255 );
  GetIndString( str255, TREK_WINDOW_STRINGS_ID, SCORE_STRING_ID );
  hStringScore = NewString( str255 );
}
```

Custom Resources

In addition to the many standard types of resources you need for your game, you will most likely also want to store and retrieve your own information. For example, Desert Trek supports ten skill levels. There are more than approximately fifty parameters that get set differently for each skill level, such as how many supplies you start the game out with, to how far the cannibals can travel during different times of the day. It would not be very elegant to set each of these variables individually depending on the skill level selected by the player. That translates to more than 500 assignment statements, adding much to the program code. A better solution would be to read this information, which does not change, at run time from the resource fork of the game.

In fact, Desert Trek uses three types of custom resources: one for the game parameters, one for the high scores, and one containing RGB values for the Mac's standard 16-color palette. The game parameters and RGB colors resources never change after I ship the game. Only the high score resource gets modified.

So, two questions naturally come to mind. How do I create custom resources for my game? How do I read and modify custom resources for my game at run time?

There are basically two ways to create custom resources for your game: manually and programmatically. If the resource contains very little data, you might as well type it in manually. For example, Desert Trek's 'clrs' resource contains 16 RGB values for colors. Since each RGB value is 6 bytes in length, there are only 96 bytes to the resource. Using ResEdit, I just created a clrs resource and typed in the data.

For more complex or lengthy resources, you will want to generate the data programmatically. Desert Trek's game parameters resources are a good example. There are a total of 10 resources of type **'GPRM'**, one for each skill level. (There are actually two additional resources of type 'GPRM' that contain the user's settings for sound and color depth warning, but we can ignore them for this example). Included with the source code for Desert Trek is the unit 'Set Parameters.c'. However, you'll notice that this unit isn't included with the Desert Trek project (in other words, it is not compiled into the game). The reason for this is that this unit was used to create the 'GPRM' resources that define the game's parameters for each skill level. You would not need this code in the final version of the game, so it's left out of the project after it's used to create the resources.

Before we go into the code to create resources, note that there's also a private resource that Desert Trek uses, which should be created before the game is distributed, but also gets modified after the game is distributed (remember, the game parameters do not change after the game is distributed). The high scores resource 'SCRS' contains the top 10 high scores for each skill level. Because they need to be modified when the user gets a high score, the code to create them was left in the final version of Desert Trek (this also allows the high scores resource to be re-created if, for any reason, it gets removed from the game).

Creating a Custom Resource Programmatically

To programmatically create resources, you need to define a handle to the data you want to store in the resource fork of the game. In other words, you can't just take any variable type and make it into a resource. After

declaring, allocating, and setting the data into a handle, use the following toolbox call to add the handle to the resource file:

```
AddResource( Handle    hData,
             ResType   ResourceType,
             short     sResourceID,
             Str255    str255ResourceName );
```

The ResType data type is defined as follows:

```
typedef unsigned long ResType;
```

Essentially, ResType is a 4-byte, or character, field. For most purposes, you will specify a four-character string, such as 'SCRS'. The following code example shows how the ResourceType parameter is specified.

Whenever your resource data changes (for example, the user gets a new high score and enters a new name to the high scores list), you most likely will want to store those changes back to the resource fork of your game. Two toolbox calls are used to set the modification state of the resource handle, and to actually force an update to the resource file:

```
ChangedResource( Handle hResource );
WriteResource( Handle hResource );
```

It's generally a good idea to mark a new resource as changed and to force a write to the resource fork of the game. The following code example shows how Desert Trek's high scores resource is created (the definition of the high scores structure and the ClearScores() function are left out of this example for brevity, but you can find the complete source code on the CD-ROM in the file **Scores Window.c**):

```
static HSCORES  hScores;

void InitializeScoresWindow( void )
{
  short  sLoop;
```

```
  // Create the high scores structure
  hScores = (HSCORES) NewHandle( sizeof( SCORES ) );
  HLock( (Handle) hScores );

  // Clear the high scores structure
  for( sLoop = 0; sLoop < 10; sLoop++ )
    ClearScores( sLoop );

  // Add the high scores structure to the resource fork
  AddResource( (Handle) hScores, 'SCRS', 128, "\pHigh Scores" );
  ChangedResource( (Handle) hScores );
  WriteResource( (Handle) hScores );
  HUnlock( (Handle) hScores );
}
```

Something you hopefully noticed was that nowhere in the previously defined toolbox calls to add, change, and write a resource, is there a parameter to tell you which resource file you want to affect. In other words, it is assumed that you are affecting the resource fork of the game itself. However, you may want to affect resource files other than the game itself. All resource calls work on what is known as the *current* resource file. Any resources created or changed will be stored in the current resource file, which defaults to the application's resource fork. The following toolbox calls allow you to create and open new resource files, as well as change the current resource file:

```
void    CreateResFile( Str255 str255ResourceFileName );
short   OpenResFile( Str255 str255ResourceFileName );
short   CurResFile();
void    UseResFile( short sResourceFileID );
void    CloseResFile( short sResourceFileID );
```

The `CreateResFile()` toolbox call may or may not create a new file. For example, you may want to create a resource fork for a file that already exists. For this case, `CreateResFile()` does not create a new file, but creates a resource fork for the already existent file. You must create a resource fork for a file before using `OpenResFile()`; otherwise the call will fail.

The following code fragment shows how you would use a resource file other than the resource fork of your game. An important thing to keep in mind is that you need to store the resource ID of your game before using `UseResFile()` to specify a different resource file. Otherwise, you won't be able to switch back after you're finished with the other resource file. The full source for this code fragment can be found in **Information Window.c.**

```
void SaveInfoWindowText( void )
{
  short   sCurrentResourceFile;
  short   sTextResourceFile;
  Str255  str255FileName = "\pMyResourceFile";

  // Save the resource id of Desert Trek itself
  sCurrentResourceFile = CurResFile();

  // Create a resource fork for str255FileName and open it
  CreateResFile( str255FileName );
  sTextResourceFile = OpenResFile( str255FileName );

  // If the open worked, continue
  if ( sTextResourceFile != -1 )
  {
    UseResFile( sTextResourceFile );

    // Put code here to read, add, and modify resources for the other file
  }

  // Close the resource file and resume using Desert Trek's resource fork
  CloseResFile( sTextResourceFile );
  UseResFile( sCurrentResourceFile );
}
```

Using a Custom Resource

Now that you've created a custom resource for your game, how do you use it? Actually, it's very simple. All you need to do is define a handle to the data structure that defines the resource you've created and use the following toolbox call:

```
Handle   GetResource( ResType   ResourceType,
                      short     sResourceID );
```

The following code fragment, which can be found in its entirety in **Scores Window.c** demonstrates the use of `GetResource()`. Notice that typecasting is used to convert the generic handle type returned by `GetResource()` to the specific handle type defined in Desert Trek.

```
static HSCORES  hScores;
static HNAME    hScoreName;

void InitializeScoresWindow( void )
{
  hScores = (HSCORES) GetResource( 'SCRS', 128 );
  hScoreName = (HNAME) GetResource( 'SCRS', 129 );
}
```

When you are finished using a resource, you need to free it from memory. The following toolbox call should be used when the resource is no longer needed:

```
ReleaseResource( Handle hResource );
```

Vegas Trek Resource Example

The CD-ROM included with this book contains a game, Vegas Trek, which was created by taking Desert Trek and changing certain resources. The idea for creating a "parody" of Desert Trek came from Mike Foley, who took Desert Trek, edited the resource fork of the game, and created Vegas Trek. Play Vegas Trek, and you'll notice that it's pretty much a completely different game (yes, all the rules are obviously the same, but all the locations, mode of transportation, events, etc. are different). This transformation was made possible by the fact that all of the graphics, menus, and text for the program are stored in the resource fork of the game. No coding changes were made (Mike didn't have the source code, and I didn't have to do anything to support his changes).

This is a great example of what can be done if a game is designed to use resources well. Just as easily, Desert Trek can be translated by anyone with the skill to do so into other languages without the need for any coding changes.

Summary

You now know what resources are, where they are stored, and how to use them. Keep in mind that creating and maintaining the resources for your game is just as important as writing the code itself. Plan to spend a significant amount of your development time creating all the resources for your game.

CHAPTER 4

WORKING WITH WINDOWS

This chapter describes how to create and use windows for your game. Windows are a program's way of displaying information to a user, such as the status of a game, a picture or view of the game, and the high scores list. Windows are also used to retrieve input from users. For example, the user may click on an object or type text into in a window. Many games create and maintain more than one window, and in some cases, can have several instances of the same type of window (similar to having several documents open in a word processor). For example, Desert Trek's help window and "Cary's Games" window are the same type of window (though they display different information), and thus use the same code for drawing the window's contents and reacting to user input.

Because the user can run several programs at the same time on a Macintosh, several windows from different applications can be visible at the same time. In addition, windows may overlap one another, obscuring portions of the overlapped window. When the user shows your game after it has been hidden or moves a window that had overlapped one of your game windows, the Macintosh operating system will generate events to your program, telling you that the window needs to be redrawn. Your program will need to respond to these events by redrawing the contents of the window. Your game itself will most likely cause its windows, or a portion of one of its windows, to be redrawn during game play even if it wasn't hidden or covered by another window. For example, when the user eats some food in Desert Trek, the status bars showing hunger and amount of food left need to be redrawn.

It is obvious that your game will need to concern itself about drawing the content of windows, and perhaps allow for some user interaction with your game through those windows. However, your game's window responsibilities don't end there. Your game will also need to allow the moving, closing, and, perhaps, sizing of windows. Routines and strategies for doing so will also be discussed in this chapter.

Anatomy of a Window

In order to understand your responsibilities concerning the maintenance of windows within your game, you first need to understand the various components of a window. Any Macintosh user surely knows the various parts of a window, but how does a programmer deal with these parts. Figure 4.1 shows the anatomy of a window.

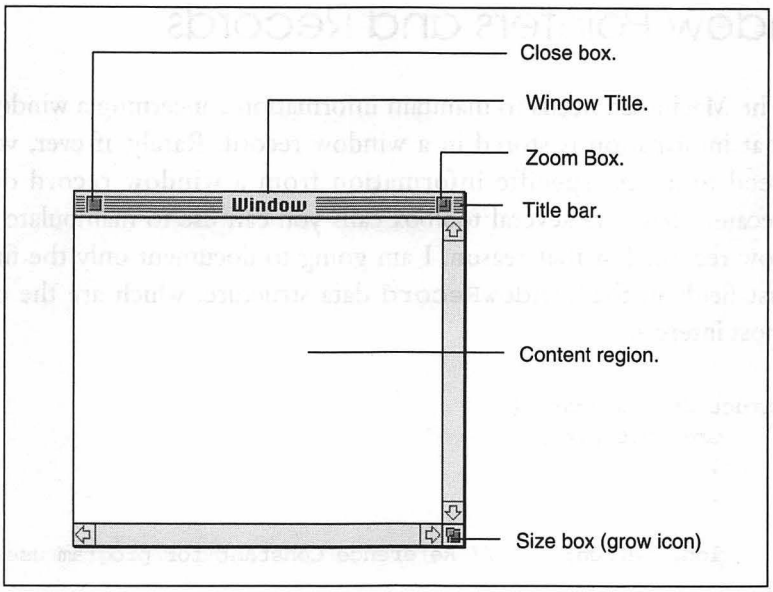


Figure 4.1 The anatomy of a window.

Chapter 2 discussed how to respond to mouse click events in the title bar, zoom box, close box, and size box. As you have seen, the Macintosh makes it fairly easy to maintain these aspects of a window, doing almost all of the work for you. Your two biggest responsibilities are to draw the content of a window and determine what to do when the user clicks the mouse in the content region of a window. The Macintosh will help you out with drawing the content region of a window by telling you when a window needs to be redrawn. Your game will also be given specific mouse click information such as which window was clicked, and the exact coordinates where the click occurred. However, before going into these topics in detail, let's examine how to load and set various properties for windows.

Window Pointers and Records

The Macintosh needs to maintain information concerning a window, and that information is stored in a window record. Rarely, if ever, will you need to access specific information from a window record directly because there are several toolbox calls you can use to manipulate a window record. For that reason, I am going to document only the first and last fields in the `WindowRecord` data structure, which are the ones of most interest:

```
struct WindowRecord {
     GrafPort port;
     .
     .
     .
     long refCon;      // Reference Constant for program use.
};

typedef struct WindowRecord WindowRecord;
typedef WindowRecord *WindowPeek;
typedef GrafPort *GrafPtr;
typedef GrafPtr WindowPtr;
```

The last field of the `WindowRecord` structure is called `refCon`, or reference constant. The reference constant is a long integer value that your game can use to store any type of information you want to be associated with that specific window. Typically, the reference constant is used by programs to differentiate different types of windows from one another (or multiple instances of the same type of window). Desert Trek has several types of windows and uses this field to give each window created by Desert Trek a unique identifier. Keep in mind that this field can be used for any purpose by your program. For example, if you would like to keep a lot of window-specific information for each window in your program, you could store a handle to the data structure describing that window information as the reference constant.

There is one additional point of interest when it comes to how Macintosh programs reference a window. In most cases, window pointers are used (`WindowPtr`) when calling toolbox functions related to win-

dows. As you can see from the previous definitions a `WindowPtr` is really just a `GrafPtr` (see Chapter 7, *Quickdraw*, for more information on the `GrafPtr` and `GrafPort` data types). Notice, too, that the first field in the `WindowRecord` data structure is of type `GrafPort`. What does this mean to you, the game programmer? Well, it means that you can use window pointers everywhere you can use graphics port pointers. In Chapter 7, you'll find out that toolbox calls used to perform drawing operations require graphics port pointers. Because you will frequently need to draw information into a window, you can use the `WindowPtr` for that window instead of always having to reference the `GrafPort` field of the window record. Believe me, this is a great convenience.

Lastly, there is a minor difference between a black-and-white window and a color capable window. Two slightly different window records are defined, one for each type. The only difference between the two window records is that a color window record (`CWindowRecord`) has as its first field a `CGrafPort` (which is a color graphics port) instead of a `GrafPort`. Your game only needs to be aware of this minor difference when it loads a window from a resource. The next section on loading windows will give you all the details.

Loading a Window

To load a window from your game's resource fork (of type 'WIND'), use the following toolbox call:

```
WindowPtr GetNewWindow(  short       sWindowResourceID,
                         void        *pStorage,
                         WindowPtr   pWindowBehind );
```

The `sWindowResourceID` parameter specifies the resource ID of the 'WIND' resource to be loaded. The `pStorage` routing specifies a pointer to an allocated block of memory large enough to hold the window record of the new window. A value of `nil` will cause the Macintosh to automatically allocate storage for the window record from the heap. You'll probably always want to use `nil` here, unless you really want to

allocate and manage the storage for the window record yourself . The `pWindowBehind` parameter specifies the plane in which your window will appear (the *Z-order*). In other words, this parameter specifies the window that will be immediately in front of the window you're loading. Most of the time you will want the loaded window to be the frontmost window on the screen. You would think that passing `nil` would accomplish this since you don't want the loaded window to go behind any window; however, you would be wrong. Passing `nil` actually causes the window to appear behind all the windows currently on the screen. To have the window appear in front of all the other windows on the screen, you need to pass -1 (typecast to type `WindowPtr`). Lastly, this toolbox call returns a window pointer to the window loaded.

Recall that I mentioned that there was a slight difference between color-capable windows and black-and-white-only windows. Well, that difference really only matters when you load the window. Another toolbox call loads a color-capable window:

```
WindowPtr GetNewCWindow(  short       sWindowResourceID,
                          void        *pStorage,
                          WindowPtr   pWindowBehind );
```

Notice that everything is the same except for the toolbox call name. The parameters and return value are the same as `GetNewWindow()`.

When you are finished with a window and want to remove it from the screen and, more importantly, remove its window record from memory, use one of the following toolbox calls:

```
// If you allocated memory for the window record yourself,
// use the following to destroy a window (the window record
// memory is not freed, so you'll need to free it yourself).
void CloseWindow( WindowPtr pWindow );

// If you had the Mac automatically allocate the window
// record (by specifying nil for the pStorage parameter of
// the GetNewWindow() call), use the following to destroy a
// window (the window record will automatically be freed).
void DisposeWindow( WindowPtr pWindow );
```

A code example showing the use of these call wills be given at the end of the section on setting window properties.

Showing and Hiding Windows

There may be times when you need to hide a window without closing it and removing it from memory. You can then later show the window without having to reload it (and without having to resize and move it, too).

```
// Hides a window.
void HideWindow( WindowPtr pWindow );

// Shows a window.
void ShowWindow( WindowPtr pWindow );
```


TIP

Because all the windows your game loads and displays will most likely need to be sized and positioned based on the size of the screen your game is running on, it is a good idea to create window resources as **not** visible. Doing so causes the window not to be displayed immediately when loaded. After loading the window, you can size and position the window and then call `ShowWindow()` to make it visible on the screen. This prevents the window from first appearing immediately after it's loaded at whatever location it's defined at in the resource fork, and then moving to its new location once you size and position it (which can be visually unappealing to the user).

Moving and Sizing Windows

The Macintosh toolbox provides the following calls to move and size windows:

```
// Moves a window to a new location on the screen. The
// horizontal (sHPosition) and vertical (sVPosition)
// values are specified in global coordinates (see the
// section on handling mouse click events in the content
// region of windows later on in the chapter for a
```

```
// description of local and global coordinates). If
// bActivate is true, the window is activated and brought
// to the front.
void MoveWindow(  WindowPtr  pWindow,
                  short      sHPosition,
                  short      sVPosition,
                  Boolean    bActivate );

// Sizes a window. The new width and height are specified in
// pixels. If bUpdate is true, the proper update events are
// send to your program to redraw any newly exposed parts of
// the window. You will almost always set this to true.
void SizeWindow(  WindowPtr  pWindow,
                  short      sWidth,
                  short      sHeight,
                  Boolean    bUpdate );

// "Zooms" a window between a zoomed-in state and a
// zoomed-out state. sPartCode is the code returned by
// FindWindow() as described in chapter 2 on toolbox basics.
// If bActivate is true, the window is activated and brought
// to the front.
void ZoomWindow(  WindowPtr  pWindow,
                  short      sPartCode,
                  Boolean    bActivate );
```

The Active Window

There can be multiple windows open on a Macintosh screen at any one time, but only one of those windows can be the active window. The active window is the window that the user is currently interacting with. For example, any keystrokes typed by the user are destined for the active window. Visually, the active window is the only window on the Macintosh desktop that has the horizontal lines drawn in the title bar (see Figure 4.2). Also notice that if the window is sizable, the grow icon is only visible for the active window (it is hidden for all nonactive windows).

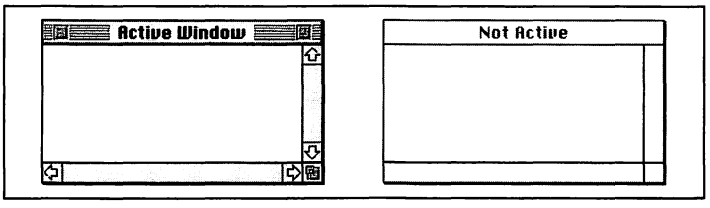


Figure 4.2 Active and inactive windows.

When an event gets posted to your game, you may need to know which game window is currently active. This is especially true when the user types from the keyboard because keyboard events posted to your game do not specify the window for which the event occurred. The following toolbox call returns the active window:

```
// Gets the currently active (front most) window.
WindowPtr FrontWindow( void );
```

There will be times when your game needs to change the active window. For example, if the user clicks on a window that is not active, you need to activate that window as well as deactivate the currently active window. The following toolbox call takes care of all of that for you:

```
// Activates the specified window and brings it to the front.
// The currently active window is deactivated.
void SelectWindow( WindowPtr pWindow );
```

If you want to manually set the highlight state of a window (in other words, draw it as active or not active), you can use the following toolbox call. Normally, you do not need to use this call because the Macintosh automatically draws the proper highlighting state when a window gets activated or deactivated.

```
// Set the highlight state for a window. Specifying true for
// bHighlight will cause the window to look active. False
// will cause the window to look not active.
void HiliteWindow(  WindowPtr  pWindow,
                    Boolean    bHighlight );
```

Lastly, remember that I briefly mentioned that the active window is the only window that has the grow icon visibly drawn. If you have a sizable window in your game, you need to manually draw the grow icon for that window when it gets activated. Use the following toolbox call.

```
// Draw the grow icon for a sizable window.
void DrawGrowIcon( WindowPtr pWindow );
```

Changing the Z-Order of Window

The z-order of windows determines which windows are on top of, or overlapping, other windows. You can specify the z-ordering of your game's windows using the following Macintosh toolbox calls. However, these calls are typically not used since they do not generate all of the appropriate activate and deactivate events. You would normally use the `SelectWindow()` toolbox call to bring a window to the front.

```
// Brings the specified window to the front.
void BringToFront( WindowPtr pWindow );

// Sends the specified window behind pBehindWindow. If
// pBehindWindow is nil, the window is moved behind all other
// windows.
void SendBehind(  WindowPtr  pWindow );
                  WindowPtr  pBehindWindow );
```

Setting Window Properties

As I stated before, there are a number of toolbox calls to manipulate the properties of a window, eliminating the need to directly access the win-

dow record itself. In reality, there are only two properties that you may want to change: the window title and the reference constant. The following toolbox calls allow you to query and set the window title and reference constant:

```
// Sets the title text for a window.
void SetWTitle(  WindowPtr  pWindow,
                 Str255     str255Title );

// Gets the title text for a window.
void GetWTitle(  WindowPtr  pWindow,
                 Str255     str255Title );

// Sets the reference constant for a window.
void SetWRefCon(  WindowPtr  pWindow,
                  long       lRefCon );

// Gets the reference constant for a window.
void GetWRefCon(  WindowPtr  pWindow,
                  long       *plRefCon );
```

Examples

The following code fragment loads a window, sizes it, sets its reference constant, and selects it as the active window. The entire function can be found in **Trek Window.c.**

```
#define TREK_WINDOW_RESOURCE_ID 128
#define TREK_WINDOW_ID           1

static WindowPtr pWindowTrekWindow = nil;
static Rect      rectTrekWindow;

static void CreateTrekWindow( void )
{
  // If this is a color Mac, use GetNewCWindow.
  if ( UsingColorGraphics( nil ) )
    pWindowTrekWindow = GetNewCWindow( TREK_WINDOW_RESOURCE_ID, nil,
                                       (WindowPtr) -1 );
```

```
    else
      pWindowTrekWindow = GetNewWindow( TREK_WINDOW_RESOURCE_ID, nil,
                                        (WindowPtr) -1 );

    // Set the reference constant so I can identify this window later.
    SetWRefCon( pWindowTrekWindow, TREK_WINDOW_ID );

    // Size the window.
    SizeWindow( pWindowTrekWindow, rectTrekWindow.right,
                rectTrekWindow.bottom, false );

    // Call my own routine to center the window on the screen.  This
    // routine can be found in "Common Functions.c".
    CenterWindow( &rectTrekWindow, &screenBits.bounds, true,
                  pWindowTrekWindow );

    // Show the window since the resource defines it as hidden.
    ShowWindow( pWindowTrekWindow );

    // Activate the window and make it font most.
    SelectWindow( pWindowTrekWindow );
}
```

Once the window is no longer needed, it needs to be disposed. The following code fragment does so (the function can be found in its entirety in Trek Window.c):

```
void DestructTrekWindow( void )
{
 DisposeWindow( pWindowTrekWindow );
}
```

At this point, you might want to go back and look at the code example given in the "Handling Keyboard Events" section in Chapter 2 to see how `FrontWindow()` and `GetWRefCon()` are used to determine which window gets a keyboard event.

Update Events

One of the most important parts of window management for your game will be drawing information into a window. When a portion of a window that was once covered by another window is uncovered, your game must redraw the part of the window that was revealed. Even if you have only one window for your game, you still need to respond to update events because windows from other applications may obscure your game window. In addition, the user may hide your game and return to it later, causing the need to redraw the entire window. Lastly, you will want to take advantage of update events to automatically cause the redraw of a window in which information has changed. Keep in mind that "information" doesn't just mean text for high scores or status for amount of food left. Information might be the position of a Klingon ship versus the players ship, or the current location of the photon torpedo on an intercept course with the players ship. The contents of a window is a snapshot of the current status of the game. That status might change 60 times a second in an arcade game with high speed animation. The actual process of drawing will be discussed in Chapter 7, *Quickdraw*, but there are a number of things that need to happen before the actual drawing occurs.

When a portion of, or an entire window, needs to be redrawn, your game receives an update event (`updateEvt`). The update event tells you which window needs to be drawn. To respond to an update event, you game would usually draw the window that needs updating. The routine that draws a window's contents is commonly referred to as the window's update routine, because you normally call it when an update event for that window is posted to your game.

How do you know what portion of the window to draw? The Macintosh actually makes this easy for you. It's simply a lot easier to redraw the whole window as opposed to figuring out what needs to be redrawn. You don't have to perform complex operations to figure out if

the player's ship belongs in the update region or not, or if the fuel indicator needs to be redrawn too. By using two toolbox calls, you can redraw the entire window, but have the Macintosh clip, or exclude, the drawing of anything that doesn't need to be redrawn. This makes it easy on your code and avoids any actual drawing that doesn't need to take place. Here are the two toolbox calls:

```
// Begin update drawing for a window.
void BeginUpdate( WindowPtr pWindow );

// End update drawing for a window.
void EndUpdate( WindowPtr pWindow );
```

So, just before drawing your window in response to an update event, call `BeginUpdate()` to have the Macintosh automatically ignore any drawing you do outside the update region. This is called *clipping*, because any drawing outside the update region is "clipped," or ignored. After drawing the window, you must call `EndUpdate()` to restore the clipping region of the window.

At times, you'll need to update a portion of the window due to a change in the information being displayed there. For example, if the user drinks in Desert Trek , I need to reflect the decrease in the water supply and player's thirst in the game window. It would be foolish to have special code to draw just the water and thirst indicators if I already have a routine to draw the entire contents of the game window to handle update events. An easy way to draw the new information is to post an update event to the game window. The event processing code in Desert Trek simply calls the window drawing routine, and the new information is drawn automatically. To post an update event to the appropriate window, you use routines to invalidate a portion of the window. When a portion of a window is marked as invalid, an update event for that window is posted to your game.

The following routines can *invalidate* or *validate* portions of a window; you invalidate portions of a window that you need to redraw and validate portions of a window that you manually draw outside a `BeginUpdate()`, `EndUpdate()` pair that you had previously invali-

dated. You only need to validate a window region if you draw something to a window before the update event had to be processed. In other words, you should rarely need to use the validate routines. Using the `BeginUpdate()`, `EndUpdate()` pair will automatically validate the entire window, so make sure to draw everything between the two calls (which you should be doing anyway), otherwise, parts of your window may not get drawn. Rectangles and regions will be discussed in Chapter 7, *Quickdraw* (the `Rect` and `RgnHandle` data types). Note that these routines do not specify which window to validate or invalidate. So, before calling one of these routines, you need to make absolutely sure that the window you want to affect is the current drawing port. The current drawing port, which will be described in detail in Chapter 7, determines where any drawing-related commands are displayed (in other words, which window gets drawn to). Use the `SetPort()` toolbox call and specify the window pointer of the window you want to validate or invalidate before making one of these calls.

```
// Invalidate a rectangular region of a window, causing an
// update event to be posted to that window.
void InvalRect( Rect *pRectInvalidate );

// Invalidate a region of a window, causing an update event
// to be posted to that window.
void InvalRgn( RgnHandle hRgnInvalidate );

// Validate a rectangular region of a window, which will be
// clipped from drawing when updating a window.
void ValidRect( Rect *pRectValidate );

// Validate a region of a window, which will be clipped from
// drawing when updating a window.
void ValidRgn( RgnHandle hRgnValidate );
```

Invalidating portions of a window is a great way to redraw windows that contain no animation, such as status windows or high score windows. However, if the window contains animation, it is generally not practical to redraw everything in the window every time something moves. Even though the Macintosh will clip all extraneous drawing, it probably would

take too much time to execute all your window drawing code just to change the position of an animated object. For example, Desert Trek, where the view fades from one part of the day to the next, does not use the traditional invalidate rectangle method because the only thing changing on the screen during the fade is the view portion of the game window. To accomplish this, I simply draw the information directly to the window without going through the invalidation and update process. This means not using `BeginUpdate()` and `EndUpdate()` when drawing the fades.

Examples

How about some code examples? For starters, the following code responds to an update event by determining which window needs to be updated and calling the appropriate routine to draw that window:

```
static void HandleUpdateEvent( EventRecord *pEvent )
{
  short      sWindowID;
  WindowPtr  pWindow = nil;

  // Check to see if the number of colors the monitor is set to changed since
  // the last update event.
  CheckMonitorColors( true );

  // Determine the window that needs updating.  Notice that I use the window's
  // reference constant to determine which drawing routine to call.
  pWindow = (WindowPtr) pEvent->message;
  sWindowID = (short) GetWRefCon( pWindow );

  // Call the appropriate routine to draw the window needing updating.
  switch ( sWindowID )
  {
    case TREK_WINDOW_ID:

      UpdateTrekWindow();
      break;

    case HELP_WINDOW_ID:

      UpdateInfoWindow( GetInfoWindowPtr( HELP_WINDOW_ID ) );
```

```
      break;

    case CARYS_GAMES_WINDOW_ID:

      UpdateInfoWindow( GetInfoWindowPtr( CARYS_GAMES_WINDOW_ID ) );
      break;

    case SCORES_WINDOW_ID:

      UpdateScoresWindow();
      break;

    case ABOUT_WINDOW_ID:

      UpdateAboutWindow();
      break;

    case APP_MODAL_DIALOG_ID:

      UpdateModalDialog( pWindow );
      break;
  }
}
```

The following routine updates the Desert Trek game window. I am using an offscreen bitmap to hold the window information and transferring it all to the screen with one toolbox call, `CopyBits()`, which will be discussed in Chapter 7.

```
void UpdateTrekWindow( void )
{
  GrafPtr  pGrafCurrent;

  // Save the current port so I set it back after drawing.  Chapter 7 on
  // quickdraw will describe this toolbox call.
  GetPort( &pGrafCurrent );

  // Set the drawing port to the Desert Trek game window.
  SetPort( pWindowTrekWindow );

  // Begin updating.  Anything outside the update region will be clipped.
  BeginUpdate( pWindowTrekWindow );
```

```
    // Draw the whole window in one shot using the CopyBits() call, which will
    // be discussed in chapter 7 on quickdraw.
    CopyBits( &bitmapTrekWindow, &pWindowTrekWindow->portBits,
              &rectTrekWindow, &rectTrekWindow, srcCopy, nil);

    // Draw the window controls (in this case, the scrollbar for the journal
    // text).  Chapter 6 on Dialogs and Controls will describe this toolbox
    // call.
    DrawControls( pWindowTrekWindow );

    // Finished drawing, so restore the clipping region for this window.
    EndUpdate( pWindowTrekWindow );

    // Restore the drawing port to whatever it was before drawing this window.
    SetPort( pGrafCurrent );
}
```

Handling Mouse Click Events in the Content Region of a Window

When the user clicks the mouse somewhere in your game window, your game receives a mouse down event (see Chapter 2 for event information). If the user clicks in the title bar, close box, zoom box, or size box of the window, you simply call a toolbox routine to automatically handle the click event. However, if the user clicks somewhere in the content region of the window, you need to handle all of the details yourself. What you need to do depends on the contents of the window. For windows that only display information, you will probably ignore the click event (other than to activate the window, if it isn't already active). For other windows, you'll need to do more. For example, Desert Trek's main window, while displaying all types of information, also contains a scrollbar control and a set of picture buttons. When the user clicks on the scrollbar or one of the buttons, something needs to happen, which brings up two points. First, you need to determine exactly where the user clicked (on the scrollbar, one of the buttons, or somewhere else). Second, you need to track the mouse click to see where the user releases the mouse. In other words, the user could click on one of the picture buttons, but change his or her mind

and move the mouse, with the button depressed, out of the picture button. Releasing the mouse button when it's not over the picture button originally clicked means that the command associated with that picture button should not be executed.

Global and Local Coordinates

To determine where the user clicked the mouse, all you need to do is look at the `where` field of the click event. The `where` field consists of a `Point` structure, which tells you the horizontal and vertical coordinates of the click. The coordinates, however, are relative to the top-left corner of the Macintosh screen. When a coordinate is relative to the top-left corner of the screen, it is referred to as a global coordinate. When a coordinate is relative to the top-left corner of the content region of a particular window, that coordinate is referred to a local coordinate. Note that this isn't relative to the top-left corner of the window itself, because that would include the title bar. Any processing you do related to mouse click events should take place in local coordinates. This is due to the fact that the user can move your window anywhere on the screen and you can't be sure how the global coordinate relates to your window unless you factor in your window's current position. You need to covert the global coordinate specified in the event record of a mouse click to a coordinate local to the window affected. It could get quite tedious querying your window's position on the screen for every mouse click event to determine where in your window the mouse was clicked. Fortunately, the Macintosh provides two routines to convert coordinates:

```
// Convert a global coordinate to a local coordinate.
void GlobalToLocal( Point &pt );

// Convert a local coordinate to a global coordinate.
void LocalToGlobal( Point &pt );
```

Again, since you do not provide a window pointer to either of these functions, you need to make sure that the correct window's port is the current graphics port by using the `SetPort()` toolbox call.

Once you determine where the user has clicked, you need to take action. The action taken depends solely on what's in your window. Again, with Desert Trek, something exciting should happen when the player clicks on certain items within its window. The first of these items is a standard scrollbar, which allows the player to scroll through the game's journal. The Macintosh toolbox provides a call that tells you whether or not a user click in the content region of a window occurred within any of the controls within that window, `FindControl()`, which will be discussed in more detail in Chapter 6, *Using Dialog Boxes and Controls*. The other item the user can click on in the Desert Trek window is any of a number of picture buttons. These picture buttons are not standard Macintosh controls, and thus must be handled separately. The rest of this chapter will show how they are handled.

First things first. How do I determine whether or not the user clicked on one of the picture buttons? If the mouse has been clicked in the Desert Trek main window, the following function from **Trek Window.c** gets called:

```
void TrekWindowMouseDown( Point pt )
{
  // Make sure the Desert Trek window is the current graphics port.
  SetPort( pWindowTrekWindow );

  // Convert the event's global coordinates to local coordinates.
  GlobalToLocal( &pt );

  // If the player did not click on the scrollbar, check to see if they
  // clicked on one of the picture command button.
  if ( !ClickedOnJournalScrollbar( pWindowTrekWindow, pt ) )
    ClickedOnCommandsButtons( pt );
}
```

If the player did not click on the scrollbar, we need to see if they clicked on a picture command button. The following routine fragment from **Trek Window.c** determines whether or not the click was in a command button using the `PtInRect()` toolbox call described in Chapter 7. Basically, the `PtInRect()` function returns true if the point in question resides within the given rectangle.

```
static Boolean ClickedOnCommandsButtons( Point pt )
{
  Boolean  bHandled = false;
  short    sLoop;

  // The sCommandButtonState variable is an array of shorts that specify
  // whether a particular command button is enabled or not. We do not process
  // mouse clicks on a command button that is disabled.
  short    *sCommandButtonState = GetCommandButtonsState();

  // In order to optimize a little, the picture command buttons rectangles
  // are divided into three groups:  those within the view rectangle, those
  // horizontally arranged under the view rectangle, and those arranged
  // vertically at the right of the screen.  We first check to see if the
  // player clicked anywhere within on of these regions before we try to
  // locate which button they clicked on.
  if ( PtInRect( pt, &rectHorizontalButtons ) ||
       PtInRect( pt, &rectVerticalButtons ) ||
       PtInRect( pt, &rectView ) )
  {
    // The player did click in one of the areas of interest, so return that
    // fact to the calling routine.
    bHandled = true;

    // Loop though every picture command button and see if that's the button
    // the player clicked on using the PtInRect() call.  In addition, that
    // picture command button must also be enabled.
    for( sLoop = NormalPaceButton; sLoop <= ExitTradingPostButton; sLoop++ )
      if ( ( PtInRect( pt, &rectCommandButtons[sLoop] ) ) &&
           ( sCommandButtonState[sLoop] == ButtonEnabled ) )

        // The user did click on an enabled picture command button.  We now
        // need to determine if the user releases the mouse button over that
        // command button.
        if ( TrackClickOnCommandButton( sLoop ) )
        {
          // The user released the mouse over that command button, so execute
          // the command (the code is left out here for brevity).
          switch ( sLoop )
          {
          }
        }
  }

  return( bHandled );
}
```

Once we determine that the player has clicked on a picture command button, we need to track the mouse as long as its button is down. This means that the picture command button needs to be drawn to reflect its state: depressed if the mouse is over the button, normal if the mouse is not over the button. In other words, the player could click on a command button (and it would have to be drawn in its depressed, or clicked on state), but then start moving the mouse around while the mouse button remains clicked. If the mouse strays out of the command button, the command button needs to be redrawn in its normal state because that command will not be selected if the user releases the mouse button. Of course, the player could be fickle and move the mouse in and out of the command button several times before making up his or her mind. We, as programmers, must allow for that case. The following routine tracks the mouse if it is clicked on one of Desert Trek's picture command buttons:

```
static Boolean TrackClickOnCommandButton( short  sCommandButton )
{
  short  sButtonState = ButtonSelected;
  short  sOldButtonState = ButtonEnabled;
  Point  pt;

  // While the mouse button is still down, track it.
  while( StillDown() )
  {
    // Get the current mouse location, which will be in local coordinates.
    GetMouse( &pt );

    // If the mouse is still over the command button they originally clicked
    // on, check to see if the button is in its enabled state (which means
    // that its drawn in the "up" position). If so, change the button state
    // so that it gets drawn in the "down" position (button selected).
    if ( PtInRect( pt, &rectCommandButtons[sCommandButton] ) )
    {
      if ( sButtonState == ButtonEnabled )
        sButtonState = ButtonSelected;
    }

    // Otherwise, if the mouse is no longer over the button they originally
    // clicked on, and if the button state is still in the "down" state
    // (button selected), set the button state to the "up" state (button
    // enabled).
```

```
    else if ( sButtonState == ButtonSelected )
      sButtonState = ButtonEnabled;

    // If the button state changed, redraw the button.  Chapter 7 on quickdraw
    // will explain the code in my DrawCommandButton() routine.
    if ( sButtonState != sOldButtonState )
      DrawCommandButton( sCommandButton, sButtonState,
                         &pWindowTrekWindow->portBits,
                         &rectCommandButtons[sCommandButton] );

    // Set the old button state to the current button state.  This way, we
    // won't bother drawing the button if its state hasn't changed.
    sOldButtonState = sButtonState;
  }

  // The user has finally released the mouse, so we need to draw the button
  // in its "up" state if it isn't already so.
  if ( sButtonState == ButtonSelected )
    DrawCommandButton( sCommandButton, ButtonEnabled,
                       &pWindowTrekWindow->portBits,
                       &rectCommandButtons[sCommandButton] );

  // Return true if the user released the mouse over the button they
  // originally clicked on.
  return( sButtonState == ButtonSelected );
}
```

CHAPTER 5

DISPLAYING AND USING MENUS

Along with windows, menus are one of the most basic components of any Macintosh application. This includes games, so even if your game takes up the entire screen, you need to provide a way to show the menu bar. This is especially true today with Multifinder and System 7 because the user may need access to any application running on the machine while playing your game (the boss screen might just not be good enough!). With this in mind, every game programmer needs to be familiar with the loading of menus, detecting when the user has selected a menu item, checking menu items to show whether or not various options are set, and maybe displaying a pop-up menu or two. This chapter will give you the rundown on menus.

Menu Bars, Menus, and Menu Items

When using proper menu terminology, it is necessary to distinguish the various parts of what we commonly refer to as *menus*. The lowest component of a menu is a *menu item*. A menu item is one entry of a menu, such as **Save..** or **Quit**. Menu items have various properties such as being enabled or disabled, checked, or having a command key equivalent. Menu items are addressed via a menu item ID, which corresponds to that item's location in a menu. For example, the first menu item has an ID of 1, the second 2, and so on.

A *menu proper* consists of a group of menu items. Your program will need to identify menus using both a menu ID and menu handle. The menu ID is specified when you create the menu in ResEdit (see Chapter 3, Resources, for full details). When the user selects a menu item, the menu ID of the menu containing the item selected will be passed to your program via an event. In addition to the menu ID, your program will need to use the menu handle of a menu in order to manipulate that menu (such as inserting, deleting, enabling, disabling, and checking menu items). You can obtain a menu's handle by using the following toolbox call and supplying its menu ID. Note that in order for the toolbox to find the menu handle with the specified ID, the menu must be loaded and inserted into the menu bar (more on that later).

```
// Obtain a menu's handle give its ID.
MenuHandle GetMHandle( short sMenuID );
```

A *menu bar* is a collection of menus. Typically, the menu bar for an application is shown at the top of the Macintosh screen. Because all applications running on the Macintosh show their menu bars at the top of the screen, only the menu bar of the currently active application is seen by the user.

It makes sense that you can manipulate menu items, such as checking a menu item, deleting a menu item, or disabling a menu item. It also make sense that you can perform operations on an entire menu at a time, such as inserting a menu into a menu bar, or changing its title. In addi-

tion, you can perform operations on the entire menu bar of an application. This includes loading menu bars and setting them to be the menu bar for your game.

There's one final point I need to make concerning menus. Before using a menu, it must be loaded from the resource fork and inserted into the menu bar. For most menus, this makes perfect sense because you'd want the menu to appear in the game's menu bar. However, there will be times when you want to use a menu that's not on the menu bar proper (at the top of the screen). Examples include hierarchical menus and pop-up menus, which will be discussed later in this chapter. These menus must also be inserted into the menu bar before you can use them. However, there's a way to do so without causing them to appear at the top of the screen in the menu bar proper, as we'll see later.

Adding the Apple Menu Items to the Apple Menu

The apple menu of every application needs to contain the apple menu items stored in the System Folder. Because these items vary from system to system based on the user's preferences, you need to add these items at run time (it is impossible to add them when creating the menu in ResEdit). The following toolbox call is used to add these items:

```
// Add the names of resources of the specified type to the
// specified menu.  All open resource files are searched,
// including the system resources.
void AddResMenu(  MenuHandle    hMenu,
                  ResType       resType);
```

Notice that this call can add items to any menu that you specify, and that you can add the names of any type of resource. For the apple menu, the resource type you need to specify to get the contents of the apple menu items folder is 'DRVR'. For system versions earlier than 7, which don't support an apple menu items folder, the 'DRVR' resource type will cause all desk accessories to be added to the specified menu. As it turns out, this

is exactly what you would want to happen—you're covered under all system versions! An example of how to use this call will be shown shortly.

Loading a 'MBAR' Resource

The convenient thing about using menu bar resources is that you can load the entire menu bar for your game in one fell swoop; well, almost, as we'll see in the example. The following toolbox call loads a menu bar resource. This routine causes all menus defined on the specified menu bar to be loaded automatically, eliminating the need to load each menu individually. Notice that the resource loaded is simply a generic handle.

```
// Load a menu bar from the resource fork.
Handle GetNewMBar( short        sMenuBarID );
```

When you are finished with a menu bar, you need to release the resource it takes in memory by calling `ReleaseResource()` (described in Chapter 3) and supplying the menu bar handle to be released.

Setting and Drawing the Menu Bar

After loading a menu bar, you need to set it as the menu bar for your game. The following toolbox call sets the menu bar and automatically inserts all menus associated with that menu bar.

```
// Sets the menu bar.
Handle SetMenuBar( Handle       hMenuBar );
```

Setting a menu bar does not draw it on the screen. You need to do so explicitly using the following toolbox call:

```
// Draws the menu bar.
void DrawMenuBar( void );
```

When you change the contents of the menu bar, you need to call `DrawMenuBar()` to reflect those changes on the screen. Changes to the menu bar include changing menu names, inserting or deleting menus, or enabling or disabling menus. You do not need to call `DrawMenuBar()` if you change a menu item.

Example Loading and Setting a Menu Bar

The following code example shows how to load a menu bar, add the apple menu items to the apple menu, and set the menu bar as the menu bar for Desert Trek (the entire function can be found in **Menus.c**).

```
#define MENU_BAR_ID    128

static Handle      hMenuBar;

static MenuHandle  hMenuApple = nil;
static MenuHandle  hMenuFile = nil;
static MenuHandle  hMenuOptions = nil;
static MenuHandle  hMenuCommands = nil;
static MenuHandle  hMenuBuy = nil;
static MenuHandle  hMenuSkillLevel = nil;

Boolean SetDesertTrekMenuBar( void )
{
  // Load the menu bar from the resource fork.
  hMenuBar = GetNewMBar( MENU_BAR_ID );

  // If the menu bar loaded okay...
  if ( hMenuBar )
  {
    // Set the Desert Trek menu bar to be the menu bar just loaded and draw
    // it on the screen.
    SetMenuBar( hMenuBar );
    DrawMenuBar();
```

```
        // Set the menu handles for each menu in the menu bar.  This will speed
        // up menu command processing in other parts of the code since we will
        // not need to obtain a menu's handle every time we try to access it.
        hMenuApple = GetMHandle( AppleMenuID );
        hMenuFile = GetMHandle( FileMenuID );
        hMenuOptions = GetMHandle( OptionsMenuID );
        hMenuCommands = GetMHandle( CommandsMenuID );
        hMenuBuy = GetMHandle( BuyMenuID );
        hMenuSkillLevel = GetMHandle( SkillLevelMenuID );

        // Add the apple menu items to the menu bar (or the desk accessories for
        // system versions previous to 7).  Note that we can do this after drawing
        // the menu bar since this command only affects the items within the menu.
        // We do not need to draw the menu bar again.
        AddResMenu( hMenuApple, 'DRVR' );
    }
}
```

Loading a Menu Resource

Sometimes you'll want to load a single menu into memory. Most of the time, it will be to support hierarchical and pop-up menus, which do not appear directly on the menu bar (and thus not defined in your menu bar resource). Pop-up menus are usually found in a window or dialog box, and in Desert Trek, you can find them in the information windows (the help window and "Cary's Games" window). The child menus (referred to as submenus) of any hierarchical menu will need to be explicitly loaded too, because they are not part of the menu bar proper. To load a menu resource, use the following toolbox call:

```
// Load a menu from the resource fork.
MenuHandle GetMenu( short sMenuResourceID );
```

After you are finished with a menu, you need to free the memory resources it takes. Use `ReleaseResource()` to do so.

Handling Menu Events

There are two methods by which menu commands can be issued by the user. The first and most obvious is for the user to select the menu command with the mouse. The second is for the user to press the command key equivalent of a menu command from the keyboard. When the user selects a menu item from one of your game menus using the mouse, your game receives a *mouse down event.* When the user types the command key equivalent of a menu command, your game receives a *key down event.* Your game must handle both these cases to properly deal with menu events.

Menu Processing For Mouse Down Events

The first thing your game does when it receives a mouse down event is to look for the window and window part in which the mouse button was clicked. This is accomplished using the `FindWindow()` toolbox call described in Chapter 2. If the window part has the value `inMenuBar`, you know that the user just clicked on a menu. Your game must then track the mouse click to determine if the user releases the mouse button over a menu item, and if so, determine what menu and menu item were selected. The Macintosh toolbox provides the following routine to automatically take care of the mouse tracking within the menu bar for you.

```
// Tracks a click in the menu bar.  This call returns the
// menu ID and menu item ID within that menu selected by the
// user.  If no menu item was selected (the user moved the
// mouse out of the menus before releasing the button), 0 is
// returned.  ptClick should be the "where" field of the
// event record (which is where the click took place).
long MenuSelect( Point ptClick );
```

Wait a minute. This toolbox call only returns one value, so how can a program determine which menu and menu item were selected by the user? Remember, you need to know both the menu ID and menu item ID

that the user selected in order to determine which menu command was selected. Why isn't the menu item ID enough? Well, the first menu item in each menu has an ID of 1. This means that the menu item ID by itself won't be enough to determine which command was selected. You'll need to know the menu ID too in order to determine the menu command selected. Looking closely at the previous toolbox call shows that it returns a long integer value. The menu IDs and menu item IDs are short integer values. `MenuSelect()` concatenates both these short integer values into one long integer value as its return code. The high order word contains the menu ID, and the low order word contains the menu item ID. The following two toolbox calls can be used to extract the low order and high order short integers from a single long integer:

```
// Get the high order short integer (word) from a long
// integer.
short HiWord( long l );

// Get the low order short integer (word) from a long
// integer.
short LoWord( long l );
```

Highlighting Menus

There's one last thing you need to do concerning menu selection. The `MenuSelect()` toolbox call automatically draws the proper highlighting of menus and menu commands as the user moves the mouse over them. However, after the user releases the mouse, it is your responsibility to unhighlight the menu that was last highlighted automatically by `MenuSelect()`. Why, do you ask, doesn't `MenuSelect()` do this for your too? After all, it goes through the trouble of tracking the mouse and highlighting and unhighlighting menus and menu items based on the mouse movement. Well, many programs do some type of processing after a menu item is selected, such as bringing up a dialog box. You'll notice that for many programs, the menu containing the menu item selected by the user remains highlighted until the command is complete (for example, the dialog box is dismissed). For this reason, it is left up to the pro-

gram itself to determine when it's appropriate to unhighlight the menu selected by the user (you can do it immediately, or wait until you've finished processing the command). The following toolbox call is used to accomplish this purpose:

```
// Highlights a menu (inverts it so that you get white text
// on a black background).  Any previously highlighted menu
// gets automatically unhighlighted.  If you specify 0 for
// the menu ID, all menus are unhighlighted.
void HiliteMenu( short sMenuID );
```

You need to call this routine at some point in your menu processing. It makes sense to put this call at the end of the processing, which is typically what most programs do. Remember to pass in a menu ID of 0 to this call in order to unhighlight all menus on the menu bar.

Desert Trek's `HandleMouseEvent()` routine, shown in Chapter 2, determines whether or not a mouse click occurred in the menu bar. The following is a code fragment from that routine so that we can see it again in a new light after learning all about menu selections with the mouse:

```
static void HandleMouseEvent( EventRecord *pEvent )
{
  short      sWindowPart;
  WindowPtr  pWindow = nil;

  // Determine the window and window part where the user clicked.
  sWindowPart = FindWindow( pEvent->where, &pWindow );

  // If the user clicked in the menu bar, call a Desert Trek routine to
  // process the menu click.  Pass that routine the menu ID and menu item ID
  // that the user selected which will be returned by MenuSelect().
  if ( sWindowPart == inMenuBar )
    HandleMenuSelection( MenuSelect( pEvent->where ) );
}
```

We will take a look at the `HandleMenuSelection()` routine in a moment, but first, let's look at how to process menu selections via the keyboard.

Menu Processing For Keyboard Events

Besides clicking on a menu using the mouse, the user can select a menu command using its command key equivalent. You can set up command key equivalents for your game's menu commands while creating the menus in ResEdit. When your game receives a `keyDown` event, you need to check to see if the command key was pressed. If so, you can assume that the user attempted a command key equivalent and process it as such. This question arises: How do you determine which menu command was selected from the keystroke? The ever helpful Macintosh toolbox provides just such a routine:

```
// Converts a keystroke into the menu ID and menu item ID
// that corresponds to that key.  The return code is the
// same as the MenuSelect() toolbox call.
long MenuKey( char chKeyPressed );
```

Desert Trek's `HandleKeyEvent()` routine, shown in Chapter 2, determines if a keystroke was a command key equivalent for a menu command. The following is an excerpt from that routine:

```
static void HandleKeyEvent( EventRecord *pEvent )
{
  // If the command key was pressed, call MenuKey(), passing it the character
  // code of the key pressed.  Send the return code from MenuKey() to Desert
  // Trek's HandleMenuSelection() routine for processing.
  if ( pEvent->modifiers & cmdKey )
    HandleMenuSelection( MenuKey( (char) ( pEvent->message &
                         charCodeMask ) ) );

}
```

Example of How to Determine which Menu was Selected

As we have seen above, both the `MenuSelect()` and `MenuKey()` toolbox routines return the menu ID and menu item ID of the menu command selected by the user. Desert Trek contains several functions

designed to determine which command to execute based on the menu ID and menu item ID returned by these toolbox routines. First, each menu in Desert Trek has its own function defined to call the appropriate command routine based on the menu item ID selected by the user (e.g., one for the File menu, one for the Apple menu). Second, there is one function that determines which menu was selected, and calls the appropriate menu handling routine to handle that selection. The following routine from **Menus.c** decodes the return code from `MenuSelect()` or `MenuKey()`, which gets passed in as a parameter, and determines which menu handling function to call:

```
void HandleMenuSelection( long lMenuSelectionInformation )
{
  short    sMenuSelected;
  short    sMenuItemSelected;

  // Break up the long which contains which menu ID and menu item ID was
  // selected by the user.
  sMenuSelected = HiWord( lMenuSelectionInformation );
  sMenuItemSelected = LoWord( lMenuSelectionInformation );

  // Based on the menu ID, call the appropriate menu handling routine, passing
  // it the menu item ID selected.
  switch ( sMenuSelected )
  {
    case AppleMenuID:

      HandleAppleMenuSelection( sMenuItemSelected );
      break;

    case FileMenuID:
      HandleFileMenuSelection( sMenuItemSelected );
      break;

    case OptionsMenuID:

       HandleOptionsMenuSelection( sMenuItemSelected );
      break;

    case SkillLevelMenuID:
```

```
            HandleSkillLevelMenuSelection( sMenuItemSelected );
            break;

        case CommandsMenuID:

            HandleCommandsMenuSelection( sMenuItemSelected );
            break;

        case BuyMenuID:

            HandleBuyMenuSelection( sMenuItemSelected );
            break;
    }

    // Unhighlight the menu selected by the user, now that processing of the
    // menu command is complete.
    HiliteMenu( 0 );
}
```

Handling Apple Menu Selections

The apple menu contains who knows what, depending on the particular Macintosh your game is running on. This begs the question; How do I know what to do when the user selects an apple menu item that isn't explicitly defined by my game? (remember, you defined at least an **About..** menu item for the apple menu). You need to do two things in order to take the appropriate action. The first is to obtain the text string contained by the apple menu item selected by the user. This is accomplished by using the `GetItem()` toolbox call, which will be described shortly. After getting the text string, you simply need to call the following toolbox routine to invoke the program, desk accessory, or show the folder pointed to by the apple menu item:

```
// Opens a desk accessory on systems earlier than 7.  Runs
// the program or shows the folder as defined by the user
// in the menu items folder for system 7 and later.  Returns
// 0 if the program or desk accessory can't be loaded.
short OpenDeskAcc( Str255 str255Name );
```

The following code from **Menus.c** shows how Desert Trek handles a menu selection from the apple menu. This function is called by `HandleMenuSelection()` which passes in the menu item ID selected by the user.

```
static void HandleAppleMenuSelection( short sMenuItemSelected )
{
  Str255   str255;
  GrafPtr  pGrafSave;

  switch ( sMenuItemSelected )
  {
    // If the About menu item was selected, load the about window.
    case AppleMenuAboutID:

      ConstructAboutWindow();
      break;

    // If the help menu item was selected, show the help window (or load
    // it if it isn't already loaded).
    case AppleMenuHelpID:

      if ( !ShowInfoWindow( HELP_WINDOW_ID ) )
        ConstructInfoWindow( HELP_WINDOW_ID, 128, 134,
                             UsingColorGraphics( nil ) );
      break;

    // If the "Cary's Games" menu item was selected, show the "Cary's Games"
    // window (or load it if it isn't already loaded).
     case AppleMenuCarysGamesID:

      if ( !ShowInfoWindow( CARYS_GAMES_WINDOW_ID ) )
        ConstructInfoWindow(  CARYS_GAMES_WINDOW_ID, 129, 135,
                              UsingColorGraphics( nil ) );
      break;

    // If an item from the apple menu items folder was selected (or, on
    // system versions earlier than 7, a desk accessory), open that item.
    default:

      // Save the current graphics port.
      GetPort( &pGrafSave );
```

```
        // Get the menu item text of the item selected by the user.  This
        // toolbox call will be discussed a little later in this chapter.
        GetItem( hMenuApple, sMenuItemSelected, str255 );

        // Open the program, folder, or desk accessory associate with that menu
        // item.
        OpenDeskAcc( str255 );

        // Set the graphics port back to what it was.
        SetPort( pGrafSave );
        break;
    }
}
```

Manually Inserting and Removing Menus from the Menu Bar

You might need to add or delete menus occasionally from the menu bar. Of course, most games probably won't need to use these functions because changing the menus on the menu bar can be disorienting to the user. Remember that you'll need to call `DrawMenuBar()` after using any of these call to visually reflect the changes made to the menu bar.

```
// Removes all menus from the menu bar. Useful for hiding
// the menu bar in full screen games.
void ClearMenuBar( void );

// Inserts a menu into the menu bar before the specified
// menu ID.  To add a menu at the end of the menu bar,
// specify 0 for sBeforeID. A value of -1 for sBeforeID has
// a special meaning, and will be described in the following
// section.
void InsertMenu(  MenuHandle    hMenu,
                  short         sBeforeID );

// Deletes a menu from the menu bar.
void DeleteMenu( short sMenuID );
```

Hierarchical Menus

Using ResEdit, you can build hierarchical menus in the menu editor by clicking on the **has Submenu** checkbox and specifying the menu resource ID of the submenu (see Figure 5.1).

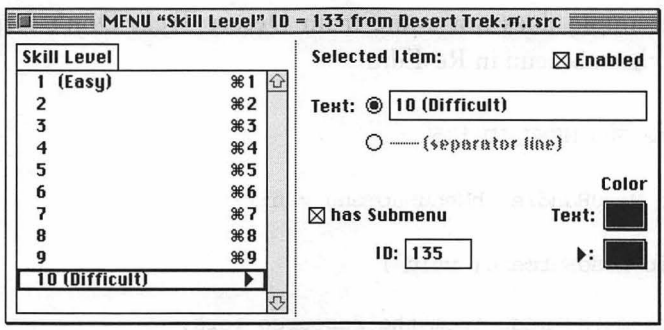


Figure 5.1 The ResEdit dialog for building menus.

A submenu is exactly the same as a regular menu (meaning that it has a menu resource ID as well as a menu ID used by your program), and getting menu selection notifications from the Macintosh operating system takes place exactly the same as normal menus (you get the menu ID and menu item ID of the menu command selected within the submenu). However, if you were to run your game, you'd notice that the submenu does not automatically appear when you select the menu item containing that submenu. Why not? Well, the answer is quite simple. You did not define that submenu in the menu bar resource because if you did so, the menu would appear on the menu bar proper. The menu won't automatically get loaded when you load the menu bar, so you need to load the menu manually using the `GetMenu()` toolbox call previously described. Loading a menu, however, isn't enough. You need to insert that menu into the menu bar before it can be used (remember, all menus must be inserted into the menu bar before they can be used). However, you need to insert the menu into the menu bar without it being shown on the

menu bar. Remember, it's only a submenu, not a main menu. To accomplish this, you need to use the `InsertMenu()` toolbox call just described, and specify as the `sBeforeID` parameter, -1. This will make the menu available for use as a hierarchical menu without being displayed on the main menu bar.

Desert Trek does not use any hierarchical menus, so the following code example does not come from Desert Trek. Don't forget to first define the submenu in ResEdit.

```
#define SUB_MENU_ID 135

static MenuHandle  hMenuSubmenu = nil;

Boolean LoadSubmenu( void )
{
  // Load the menu from the resource fork.
  hMenuSubmenu = GetMenu( SUBMENU_ID );

  // Insert the menu into the menu bar so that it's not actually
  // shown on the menu bar proper, but is available for use as a
  // submenu.
  InsertMenu( hMenuSubmenu, -1 );
}
```

Pop-up Menus

Pop-up menus don't appear on the menu bar proper, but can be "popped up" just about anywhere on the screen. Desert Trek uses pop-up menus in the help dialog box and "Cary's Games" dialog box. Just like hierarchical menus, pop-up menus need to be manually loaded using `GetMenu()`, and inserted invisibly into the menu bar using `InsertMenu()` with the `sBeforeID` parameter set to -1. To pop-up a menu anywhere on the screen, use the following toolbox call:

```
// Causes a popup menu to be displayed on the screen.  The
// menu must first be loaded and inserted into the menu bar
// (with sBeforeID = -1), and the coordinates are specified
// in global coordinates.  sPopupItem denotes the menu item
```

```
// to be selected when the popup menu appears (0 means no
// item will be selected).
long PopupMenuSelect( MenuHandle   hMenuPopup,
                      short        sTop,
                      short        sLeft,
                      short        sPopupItem );
```

The `sPopupItem` parameter specifies which menu item within the pop-up menu will the currently selected item when the menu pops up. In other words, when the menu pops up, the menu item with the ID of `sPopupItem` will be highlighted first. The two coordinates specified are the top-left corner, in global coordinates, of the location of the selected menu item. This is done so that when the pop-up menu appears, the mouse is positioned over the menu item selected. If there is no selected menu item (`sPopupItem` is 0), the coordinate specified are at the top-left corner of the pop-up menu itself. If a menu item is selected, menu items located before the selected item will be drawn above these coordinates, and menu items located after the selected menu item will be drawn below these coordinates.

`PopupMenuSelect()` returns the same value as `MenuSelect()` or `MenuKey()`. It is a long integer value whose high order word contains the menu id of the menu selected (which we already know because the menu handle corresponding to that ID was passed in as the first parameter to this toolbox call), and low order word contains the menu item ID of the item selected (0 if no item is selected).

Pop-up menus aren't as easy to maintain as normal menus because you need to manually draw the location of the pop-up menu on the screen and determine for yourself when the user has clicked on that pop-up menu. When I say draw the location of the pop-up menu, I mean the location in your window where the user can click to make the pop-up menu to appear. This is typically an outlined box with a shadow containing some type of text (see Figure 5-2 showing Desert Trek's help window). Because we have not yet covered drawing routines, I will point you to the `DrawTopicsMenu()` function defined in the Information Window.c module of Desert Trek. After reading more on drawing, take a look at this function to see how you'd draw the box and shadow of a pop-up menu location in your own window. The drawing of the pop-up up

menu in its popped up state, including all highlighting and mouse tracking, is taken care of by the Macintosh.

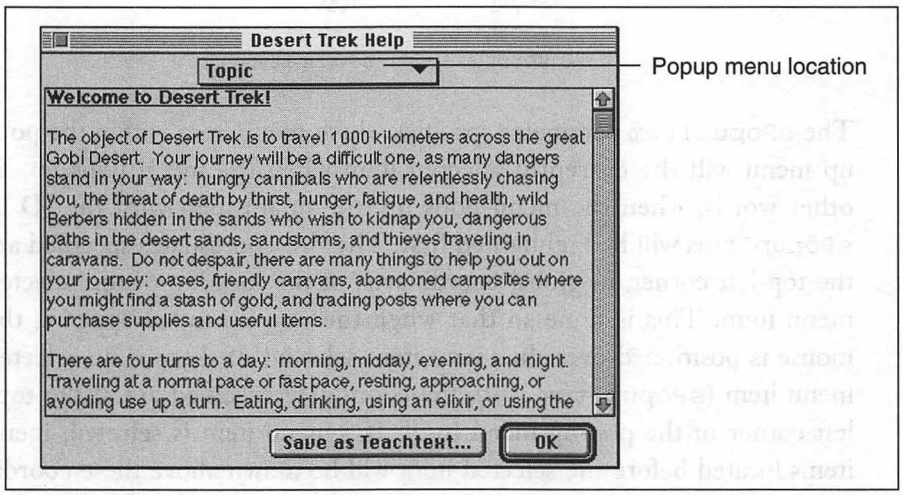


Figure 5.2 The Help window from Desert Trek.

To determine when to pop up a pop-up menu, you need to process the mouse down event in the content region of the window containing the pop-up menu. Simply check to see if the mouse was clicked within the rectangle bounding the pop-up menu. If so, pop up the menu. Remember that the pop-up menu coordinates are specified in global coordinates, not local.

Since Desert Trek's pop-up menus appear in a dialog box, I use a user-defined control to define the pop-up menu's location. This allows me to treat the pop-up menu as a standard control, which has the advantage of telling me when it has been clicked on (so I don't need to check mouse coordinates myself). Chapter 6 on using dialog boxes and controls will discuss this trick in more detail.

Inserting and Deleting Menu Items

You may occasionally need to add or delete items from menus. The following toolbox calls allow you to do so. Take caution, however, because if you insert or delete items from a menu, you will be changing the item

IDs of all menu items located after the item that was inserted or deleted. Make sure to account for that in the part of your code, which determines the menu item selected by the user.

```
// Add a menu item to the end of a menu.
void AppendMenu(  MenuHandle  hMenu,
                  Str255      str255Definition );

// Insert a menu item after the specified location.  If
// sAfterItemID is 0, the item will be inserted at the
// beginning of the menu.
void InsertMenu(  MenuHandle  hMenu,
                  Str255      str255Definition,
                  short       sAfterItemID );

// Delete the specified menu item from a menu.
void DelMenuItem( MenuHandle  hMenu,
                  short       sItemID );

// Returns the number of menu items in a menu.
short CountMItems( MenuHandle hMenu );
```

Notice that the strings taken by the `AppendMenu()` and `InsertMenu()` toolbox calls are called definition strings. The reason they are called definition strings is that they specify more than just the text of the item. The following modification characters can be used within the definition string:

Character	Meaning
;	Item separator. Used to insert more than one item at a time
(	Disable the item
/	Specify a command key equivalent
<	Text style. B=Bold, I=Italics, U=Underline, O=Outline, S=Shadow
!	Mark the item
^	Specify an icon

For example, if you want to associate a command key equivalent to an inserted menu item, you need to append the / character and the command key equivalent to the menu item string. To specify a command equivalent of command-S for a menu item titled **Save**, the definition string would be **"Save/S"**.

Getting and Setting a Menu Items Text

The following two toolbox calls allow you to retrieve or set the text of a menu item. In order to support the apple menu, you need to use `GetItem()` to obtain the apple menu item's text before calling `OpenDeskAcc()` to run the program associated with that menu item.

```
// Get a menu item's text.
void GetItem( MenuHandle     hMenu,
              short          sMenuItemID,
              Str255         str255ItemText );

// Set a menu item's text.
void SetItem( MenuHandle     hMenu,
              short          sMenuItemID,
              Str255         str255ItemText );
```

Note that the modifier characters described in the previous section on inserting and deleting menu items have no effect here. In fact, if you want to use one of the character modifiers in the actual text of a menu item, you must use `SetItem()`.

Enabling and Disabling Menus and Menu Items

Frequently you will need to enable or disable certain menu items during game play. Some commands may not make sense during certain points in the game. For example, in Desert Trek you can't buy any supplies unless you're in a trading post. This means that the Buy menu and all its items must be disabled

when the player isn't in a trading post, but they must be enabled when a player enters a trading post. In fact, it's even a little more complicated than that. Even when a player is at a trading post, they may not have enough money to purchase a certain item. In that case some of the menu items in the Buy menu will need to be disabled (the player doesn't have enough money to buy the item), and other items will need to be enabled (the player has enough money). This logic applies to almost every menu command in Desert Trek. It is generally a good idea to create a single routine that enables and disables all menu items based on the current game conditions. You should call this routine after anything happens that might affect the state of the menus. Desert Trek and every other game I've written call such a routine after every command issued by the game player. If you want to look at what this routine for Desert Trek, see the `AdjustMenus()` function (which in turn calls `AdjustCommandsMenu()`, `AdjustBuyMenu()`, and `AdjustFileMenu()`) in the unit Menus.h. I'd put it here, but it's quite long.

The following toolbox calls enable and disable menu items.

```
// Disable a menu item.  To disable the whole menu, specify
// 0 for sMenuItemID.
void DisableItem(  MenuHandle  hMenu,
                   short       sMenuItemID );

// Enables a menu item.  To enable the whole menu, specify
// 0 for sMenuItemID.
void EnableItem(   MenuHandle  hMenu,
                   short       sMenuItemID );
```

To enable or disable an entire menu, specify 0 as the menu item id (`sMenuItemID`). Remember, to have an enabled or disabled menu appear correctly on the menu bar, you need to call `DrawMenuBar()`. You do not need to call `DrawMenuBar()` if you are just enabling or disabling a menu item.

Checking Menu Items

You will often need to check a menu item, especially for options that can be either on or off. For example, Desert trek has a menu item under the

Options menu for **Sound**. Selecting the menu item will turn sound on or off. When sound is on, the menu item is checked to provide a visual clue to the game player that sound is on (just in case the sounds emitted by the computer aren't clue enough!). The following toolbox call checks or unchecks a menu item.

```
// Sets the check state for a menu item.  A value of true
// for bChecked checks the item.  False unchecks the item.
void CheckItem(    MenuHandle    hMenu,
                   short         sMenuItemID,
                   Boolean       bChecked );
```

CHAPTER 6

USING DIALOG BOXES AND CONTROLS

Dialog boxes and controls are essential elements of any Macintosh application, including games. Dialog boxes provide a way for programs to interact with the user. Frequently, a game will display a dialog box to ask for information from the user such as a name for a high scores list or the keyboard setting for moving the player's ship. Dialog boxes are also a convenient way to display certain types of information such as help screens and high scores lists. Controls are elements that can reside in windows or dialog boxes and are the actual objects with which the user interacts. Standard controls include push buttons, radio buttons, check boxes, and scroll bars.

This chapter will show you how to use dialog boxes and controls in your game. Controls will be discussed first, because they play an important part in dialog boxes.

Controls

Types of Controls

The Macintosh operating system provides several types of controls, which are often referred to as the standard controls. In addition to the standard controls, the control manager allows programmers to create other types of controls to suite their needs. However, the creation and use of so-called user-defined controls goes beyond the scope of this book. Don't fret, though, because the standard controls provide enough functionality to cover all your game's needs.

The standard controls include buttons and scroll bars. Button controls are really a class of controls that include push buttons, radio buttons, and check boxes. Every Macintosh user is familiar with these controls, and your game should take advantage of them when prompting the user for information (see Figure 6.1).

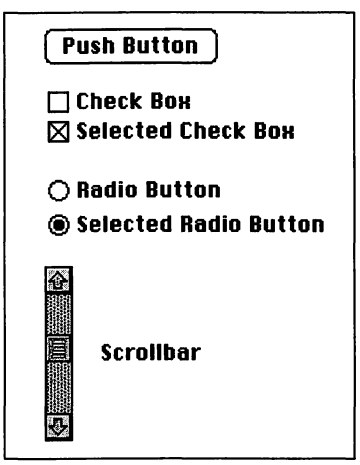


Figure 6.1 The standard controls.

Push buttons are used to allows users to complete the entering of data, be it saving it by pressing **OK**, or discarding it by pressing **Cancel**. They are also used to execute commands and bring up other dialog boxes for entering additional types of information. Last, they are typically the method by which the user dismisses dialog boxes and alerts. Radio buttons are used when the user is allowed to choose one option from a list of options. Normally arranged in groups (you would never have a single radio button appear by itself in a dialog box or window because that simply wouldn't make sense), exactly one radio button in that group is selected at any one time (no more, no less). The others radio buttons within that group are deselected. Check boxes are used when the user is given an option that can be yes or no, or on or off. They can appear singly or in groups. Finally, scroll bars are used to allow the user to see more information than can comfortably fit in the dialog box or window. Though primarily used for scrolling text, scroll bars are also useful for scrolling other large game elements such as maps.

Control Parts

Controls have distinct parts, and the user can affect controls in different ways using the different parts. Button controls have only one part, the button itself (the text of a radio button and check box is considered part of the button), so clicking anywhere on a button control has just one effect: selecting, or in the case of check boxes, possibly deselecting that button. Scroll bars, on the other hand, have several parts: the up button, down button, page up region, page down region, and thumb parts (see Figure 6.2). When responding to control events for scroll bars, you will need to know what part of the control was clicked on by the user in order to determine what to do (e.g., scroll the text up or down, scroll by just one line or a whole page at a time, etc.). The following toolbox constants define the various parts of a scroll bar control (we will shortly discuss how your game can determine what part of a scroll bar was clicked on by the user).

```
enum {
        inUpButton = 20,
        inDownButton = 21,
        inPageUp = 22,
        inPageDown = 23,
        inThumb = 129,
};
```

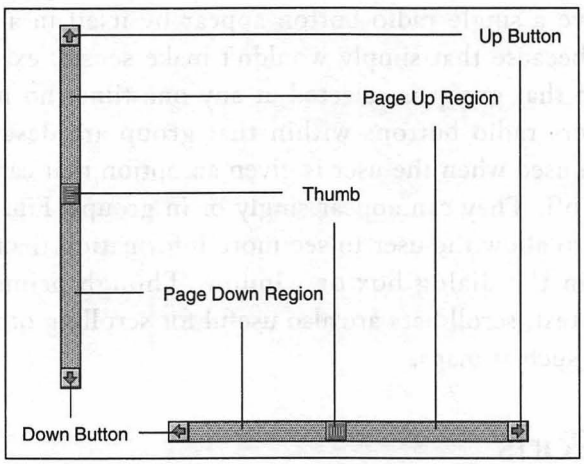


Figure 6.2 The parts of a scroll bar.

Control Records and Control Handles

We learned in Chapter 4 on windows that each window your game creates has a window record associated with it which contains the pertinent information about that window. Similarly, controls created by your game have a control record associated with them which contains all the information associated with that control. Again, your game should not directly access the fields of a control record directly. Instead, you should use one of the many toolbox calls that give you access to the control record information. The toolbox routines that set and retrieve a control's properties will be given in the following sections.

The control record contains many properties to define how a control looks on screen as well as how the user is allowed to interact with it. Though I'm not going to show you the specific fields contained within the control structure, I'll discuss the various properties in detail. When you load a control from a resource fork, the handle to the control record will be returned to your game. This control handle will be used to access the various fields of the control. The following toolbox definitions define the control record and handle (I'm leaving out most of the control record fields because you can access them using toolbox routines).

```
struct ControlRecord {
      .
      .
      .
    long    contrlRfCon;
    Str255 contrlTitle;
};

typedef struct ControlRecord ControlRecord;
typedef ControlRecord *ControlPtr, **ControlHandle;
```

The following properties are defined in a control record.

1. **Control Owner.** Each control can have only one owner. The owner of a control is the window or dialog box that contains that control. You specify the owner when a control is loaded.
2. **Control Rectangle.** The control rectangle defines where, in local coordinates, the control resides within a window or dialog box. You specify a control's initial location when defining it in ResEdit, but you can change its location using a toolbox call discussed later on.
3. **Control Visibility and Highlight State.** Controls can be either visible or invisible. Only visible controls are drawn on the screen and can be interacted with by the user. A control's highlight state affects how that control gets drawn on the screen when visible. The most common use of a control's highlight state is to enable and disable the control. A disabled control cannot be manipulated by the user, and it's typically drawn in an inactive state ("grayed out").

4. **Control Values.** Control values have two major purposes. The first and foremost is to provide the control's drawing procedure information on how to draw the control. When the user manipulates a control, the results of that manipulation generally cause the value of the control to change. For example, if the user clicks on a check box, the check within that box toggles on and off. When the user clicks in a scroll bar, the thumb part of the scroll bar moves up or down (or left or right) to reflect the action the user has taken. The value of the control determines how certain parts of that control are drawn (checked or unchecked, thumb location, etc.). This means that your game can influence how a control gets drawn on the screen by setting its value.

   The second purpose of a control's value is to provide a way for your program to determine the state of a control, and act accordingly. For example, you can determine whether or not a check box is checked by reading its control value, and set or reset a game option dependent on that check box . By reading a scroll bar's value, you can scroll the text associated with that scroll bar to the desired location.

   Controls have three values associated with them: the minimum value, maximum value, and current value. For button controls (radio buttons and check boxes), the minimum control value is 0 and the maximum control value is 1. You won't need to change these default values because it only makes sense for buttons to have one of two states: checked or unchecked (or on or off for radio buttons). A scroll bar's minimum value is typically 0 (for simplicity), and the maximum value is generally determined by the program, depending on the amount of information that can be scrolled. A scroll bar's maximum value would be set such that when the scroll bar's current value is equal to the maximum value, the user would be viewing the end of the information controlled by the scroll bar. The scroll bar's drawing routine uses all three control values to determine where to draw the thumb of the control.

5. **Control Reference Constant.** The control structure provides a long integer field that can be used by your game to store any control specific information. This is very similar to the reference constant windows contain for your game's use, and can be used to distinguish between multiple controls within a window.

6. **Control Title.** The last field of a control structure is the control's title. The control's title determines what text gets drawn on the screen in or near that control. The title text for push buttons gets drawn within the push button. The title text for radio buttons and check boxes gets drawn just to the right of the control. A scroll bar's title text does not get drawn.

Creating, Loading, and Destroying Controls

Controls can be defined in the resource fork of your game using ResEdit. You can then load these controls into any window your game maintains. Alternatively, you can create a control from scratch using a toolbox routine. Desert Trek creates a standard scroll bar control for use on the main game screen. The player uses the scroll bar control to scroll the game's journal text. The following toolbox routines load or create a control into the specified window. Note that, unlike the `GetNewWindow()` toolbox call, you are not given the opportunity to manage the memory needed by a control record. Why? I don't know, so just deal with it (you aren't really trying to manage window record memory by yourself anyway, right?).

```
// Get a control from the resource fork.
ControlHandle GetNewControl(  short      sResourceID,
                              WindowPtr  pWindow );

// Creates a control from "scratch".
ControlHandle NewControl(  WindowPtr     pWindow,
                           Rect          *pRect,
                           Str255        str255Title,
                           Boolean       bVisible,
                           short         sInitialValue,
                           short         sMinimumValue,
                           short         sMaximumValue,
                           short         sControlType,
                           long          lReferenceConstant );
```

The following control types are defined to allow you to create any standard control with the `NewControl()` toolbox routine. Specify one of them as the `sControlType` parameter.

```
enum {
    pushButProc = 0,        // Standard pushbutton
    check boxProc = 1,      // Standard check box
    radioButProc = 2,       // Standard radio button
    scroll barProc = 16,    // Standard scroll bar
}
```

Once you are finished with a control, you need to remove it from the screen and free up the memory taken by its control record. The following toolbox calls remove either one specific control, or all controls for a particular window. Note that when you call `CloseWindow()` or `DisposeWindow()` to remove a window from the screen and its window record from memory, all the controls associated with that window are automatically disposed of for you. So, you only really need to use these calls if you plan on removing a control from a window that will continue to be displayed, something you probably aren't going to need to do all that often for normal games.

```
// Destroys a control, removing it from the screen.
void DisposeControl( ControlHandle hControl );

// Removes all controls contained within a window.
void KillControls( WindowPtr pWindow );
```

Moving and Sizing Controls

Typically, you specify a control's location and position when you define that control in ResEdit, or create it using `NewControl()`. Occasionally, however, you may want to move or size a control within your code after the control has been created. The following toolbox calls move and size controls within your windows and dialog boxes. Note, however, that in addition to using these calls, you must call the `SetDItem()` toolbox call to have your changes reflected onscreen when moving controls within dialog boxes. This routine as well as an example of `MoveControl()` will appear in the dialog boxes section of this chapter.

```
// Changes a control's location within a window or dialog.
// The coordinates specified are local coordinates.
void MoveControl(  ControlHandle  hControl,
                   short          sHPosition,
                   short          sVPosition );

// Changes a control's size (width and height).
void SizeControl(  ControlHandle  hControl,
                   short          sWidth,
                   short          sHeight );
```

Showing, Hiding, and Drawing Controls

You may occasionally want to hide controls. Be careful, though, because it almost always makes more sense to the user to have a control drawn in its disabled state instead of being hidden. The following toolbox routines show and hide controls.

```
// Hides a control.
void HideControl( ControlHandle hControl );

// Shows a control.
void ShowControl( ControlHandle hControl );
```

When your window receives an update event, you typically respond by drawing that window's contents. If your window contains controls, those controls will need to be drawn too. The following two toolbox routines cause a window's controls to be redrawn.

```
// Draws all the controls for a given window.  Call this
// routine in your window's update procedure if that window
// contains any controls.
void DrawControls( WindowPtr pWindow );

// Draws only those controls for a window that fall within
// the region specified.  This routine is more efficient than
// DrawControls() since it skips the drawing of controls
// outside the specified region.  However, you'll have to
// determine that region yourself, so it's not as easy to
```

```
// use.
void UpdtControls(  WindowPtr  pWindow,
                    RgnHandle  hRegion );
```

Changing a Control's Highlight State

The following toolbox call allows you to change a control's highlight state. The highlight state of a control is a short integer value from 0 to 255, where 0 means no highlighting (the normal active state for a control) and 255 means that the control is disabled (inactive). For some historical reasons, the value of 254 is considered invalid. Values of 1 through 253 define which part of a control is highlighted. Typically, though, you will only use highlight state values of 0 and 255 to enable or disable controls within your dialog boxes and windows. A disabled control can't be interacted with by the user, and thus will generate no events to your game if clicked on.

```
// Set the highlight state for a control.  0 means enabled,
// 255 means disabled.
void HiliteControl(  ControlHandle  hControl,
                     short          sHiliteState );
```


TIP

Many of the push buttons in Desert Trek's dialog boxes allow the user to activate ("push") them via the keyboard instead of having to click on them with the mouse. For example, the Return and Enter keys always perform the same action as clicking on the default push button in any dialog box (the default push button is drawn with a border to denote it as such). In addition, the Escape key can be used to cancel any dialog boxes that contain a Cancel button (or, in the case of the dialog box used to enter a player's name into the high scores list, the **Not Interested** button, which is pretty much the same thing as a **Cancel** button). In order to provide visual feedback to the user when using the keyboard to "click" on a button, I set the push button's highlight state to cause the button to be automatically redrawn on the screen first in the "pushed" position, then back in the normal position. This causes the button to look like it had been clicked on with the mouse. It's a neat effect, and what makes it so neat is that the user probably doesn't even think twice about it. They just say, "Yeah, it's supposed to work that way. No big deal."

```
void AnimateButton( DialogPtr  pDialog,
                    short      sItemID )
{
  ControlHandle hControl;

  // Get the control's handle.  Note that this is a Desert Trek routine, not
  // a toolbox routine.  This routine will be discussed in the section on
  // dialog boxes.
  hControl = GetItemHandle( pDialog, sItemID );

  // Cause the button to be drawn in it's "pushed" state.
  HiliteControl( hControl, 1 );

  // Give the user some time to see which button they caused to be "pushed".
  NiceDelay( 8 );

  // Cause the button to be drawn back in its normal state.
  HiliteControl( hControl, 0 );
}
```

Changing Control Values

We learned that controls have a minimum, maximum, and current value. Of course, the Macintosh toolbox provides a set of routines to query and set these values. Note that a control's current value is not allowed to stray outside the minimum and maximum values. If you try setting a control's current value greater than the maximum value, the current value will be set to the maximum value. If you try to set a control's current value lower than the minimum value, the control's current value will be set to its minimum value. As an aside, this is one excellent reason why you shouldn't directly manipulate a control record's value yourself. You might accidentally set a control value outside that valid range, and who knows what kind of dire consequences might happen when the poor, confused control attempts to draw itself.

```
// Set the current value for a control.
void SetCtlValue(  ControlHandle  hControl,
                   short          sValue );

// Set the minimum value for a control.
void SetCtlMin(  ControlHandle  hControl,
```

```
                short           sValue );

// Set the maximum value for a control.
void SetCtlMax(  ControlHandle  hControl,
                 short          sValue );

// Get the current value for a control.
short GetCtlValue( ControlHandle hControl );

// Get the minimum value for a control.
short GetCtlMin( ControlHandle hControl );

// Get the maximum value for a control.
short GetCtlMax( ControlHandle hControl );
```

Changing Control Properties

A control's title and reference constant can be set or retrieved using the following toolbox routines.

```
// Set the title text of a control.
void SetCTitle(  ControlHandle  hControl,
                 Str255         str255Title );

// Get the title text of a control.
void GetCTitle(  ControlHandle  hControl,
                 Str255         str255Title );

// Set the reference constant for a control.
void SetCRefCon(  ControlHandle  hControl,
                  long           lRefCon );

// Get the reference constant for a control.
void GetCRefCon(  ControlHandle  hControl,
                  long           *plRefCon );
```

Determining Which Control Was Clicked

Now that you've loaded all your controls and set their properties just right, how do you tell when the user has clicked on one of them? In addition, what does your game have to do in response to a user's click in one of your controls? Remember, when the user clicks anywhere in your window, your game gets a mouse down event specifying the content region of that window. If that particular window has any controls, you need to determine if one of them was clicked on in your routine handling mouse down events in the content region. The following toolbox routine will tell you if a control within your window was clicked on by the user.

```
// Determines which control within a window was clicked on
// by the user.
short FindControl(  Point          pt,
                    WindowPtr      pWindow,
                    ControlHandle  *hControl );
```

This is a fairly busy toolbox call, taking and returning several values. Let's take a closer look at its parameters and return code. The input parameters to this function are the window pointer of the window clicked on, and the point clicked within that window. The `pt` parameter is specified in local coordinates, so you will need to convert from the global coordinates given to you in the `where` field of the event record for the click event. The function also takes a pointer to a control handle, which gets the control handle of the control clicked on. If no control is clicked on, the `hControl` parameter is set to `nil`. Use this value to determine if a control within the window was clicked on by the user. Last, this toolbox call returns the part of the control clicked on by the user (assuming, of course, that a control was actually clicked on). Shortly we will see an example of handling scroll bar clicks where the control part clicked on by the user is important to processing that click.

After the user clicks on a control, what happens? As with most mouse clicks, we need to track the mouse to see whether or not the user releases the mouse button in that control. In addition, the control needs to be drawn appropriately to reflect the location of the mouse while the mouse button is down. Fortunately, the Macintosh provides a toolbox call to do just this.

```
// Tracks the mouse when the user clicks on a control.  pt
// is specified in local coordinates, and should be the same
// value specified in FindControl().
void TrackControl(  ControlHandle  hControl,
                    Point          pt,
                    ProcPtr        pProcAction );
```

There's one very interesting argument to this toolbox call, and that's `pProcAction`. This parameter specifies a *callback function* that you define to perform any actions while the mouse is clicked down on a control. A callback function is one of your game's own functions that you want a Macintosh toolbox routine to call. In this case, the callback function is called periodically while the mouse button is down in the control being tracked. You can most certainly specify `nil` as the `pProcAction` parameter if you don't want to take any special action while the mouse button is down. In fact, there is usually only one case where you actually want to specify this parameter. That's the case where the user clicks in a scroll bar part other than its thumb. You typically don't want to take any action when the user clicks and drags the thumb of a scroll bar since you won't redraw the text (or graphics) being scrolled until the user drops the thumb by releasing the mouse button. In the case where the user clicks one of the scroll bar's other parts, you'll want to continually update the information being viewed to reflect the scroll bar's new value. For example, if the user clicks in the up arrow of a scroll bar, the text associated with that scroll bar should be scrolled one line up. As long as the user keeps the mouse button clicked in the up arrow, you need to keep scrolling the text up, one line at a time. In order to do so, you must specify a callback routine when calling `TrackControl()`. Otherwise, your program won't get control until the user releases the mouse button, and the user won't see the text being scrolled while the mouse button is down.

The callback function you specify must be declared as `pascal`, because the Macintosh toolbox uses the pascal calling convention (see Chapter 2 on toolbox basics if you need a refresher). In addition, your callback function needs to take two parameters: a control handle and control part. You will need these values in order to determine how to draw any changes taking place due to the mouse down event in that control. Your function should look like the following function prototype.

```
pascal void MyCallback(  ControlHandle  hControl,
                         short          sControlPart );
```

We'll see exactly how to use this callback function in the following example of how to handle scroll bar control events.

Scroll Bar Example

Since the scroll bar is the most difficult standard control to use, and the most versatile, let's look at how Desert Trek handles the journal scroll bar in the main game window. The journal scroll bar allows the player to scroll the journal's text, which is a log of all the events that take place during their trek across the desert. This example will concern itself with the scroll bar only (loading the scroll bar, setting its values, and responding to click events), and not the actual drawing of the text effected by the scroll bar. See Chapter 8, *Incorporating Text*, for specific details on drawing the text itself.

Desert Trek creates the journal scroll bar from scratch using the `NewControl()` routine. The following code excerpt comes from **Journal.c**.

```
static ControlHandle  hControlJournalScroll bar = nil;

void ConstructJournal( WindowPtr  pWindowTrekWindow,
                       Rect       *prectJournalScroll bar,
                       Rect       *prectJournal )
{
  // Create the journal scroll bar.  The rectangle specified was computed so
  // that the scroll bar appears just to the right of the textedit field
```

```
    // containing the journal text.
    hControlJournalScroll bar = NewControl( pWindowTrekWindow,      // Owner
                                            prectJournalScrollbar,  // Rectangle
                                            "\p",                   // No Title
                                            true,                   // Visible
                                            0,                      // Value
                                            0,                      // Min Value
                                            0,                      // Max Value
                                            scrollbarProc,          // Scrollbar
                                            0 );                    // RefCon
}
```

As you can see, the control values are initialized to 0 because there is no journal text when the scroll bar is created (this happens before the first game even begins). As I stated above, the control values are one of the most important aspects of a scroll bar control, because they determine how the scroll bar is drawn. This not only includes whether the scroll bar is active or not, but the positioning of the thumb. Remember that a scroll bar's thumb location reflects the proportion of text that can be scrolled up or down (in other words, where in the text the user is currently viewing). For example, if the user is looking at line 75 out of 100, the scroll bar thumb should be drawn about three quarters of the way down from the top of the scroll bar.

The question arises, what should be scroll bar's control values? Well, the minimum value should always be 0, to provide a base starting point. What should the maximum value be? That depends on how many lines of text are contained in the journal, as well as the number of lines of text that can be displayed on the screen at one time. The scroll bar's maximum control value needs to reflect the number of lines of text that can actually be scrolled. The number of lines that can be scrolled is equal to the total number of lines in the journal minus the number of lines that fit on the screen. For example, if the journal contains six lines of text, and four lines of text can be displayed on the screen at a time, the scroll bar's maximum value needs to be 6 - 4, or 2. This makes perfect sense if you think about it. If four lines are displayed on the screen, the user needs to click the down arrow twice in order to see the last line of text The first click on the down arrow will scroll the text up one line, making line 5 visible. A second click on the down arrow will make line 6 visible, and since there's no more text, the scroll bar should now be at it's maximum value.

It took two clicks to scroll through all the text, so the scroll bar's maximum control value needs to be 2. The following code computes and sets the scroll bar's maximum value whenever text is added to the journal (just the code directly related to the scroll bar is shown).

```
static short sLinesPerPage;

void AddJournalText( char    *szText,
                     short    sLength,
                     Boolean  bUpdateWindowNow )
{
  short    sMaximumScrollValue = 0;

  // The scroll bar's maximum control value needs to be the total number of
  // lines in the journal minus the number of lines that can be displayed on
  // the screen at one time.
  sMaximumScrollValue = (*teJournal)->nLines - sLinesPerPage;

  // If all of the journal text fits on the screen, there is nothing to
  // scroll, and the maximum value is 0.
  if ( sMaximumScrollValue < 0 )
    sMaximumScrollValue = 0;

  // Set the scroll bar's highlight state (active or inactive) based on
  // whether there's something to scroll (if there isn't anything to scroll,
  // the scroll bar needs to be disabled).
  HiliteControl( hControlJournalScrollbar, sMaximumScrollValue ? 0 : 255 );

  // Set the scroll bar's maximum control value.
  SetCtlMax( hControlJournalScrollbar, sMaximumScrollValue );

  // Set the scroll bar's current value to be the same as the maximum value.
  // This will force the journal text to scroll to the end every time new
  // text is added.  This is done so the user can see the new text
  // immediately without having to scroll down.
  SetCtlValue( hControlJournalScrollbar, sMaximumScrollValue );

  // If the screen is to be updated, synchronize the text displayed with the
  // current value of the scroll bar.  In other words, actually scroll the text
  // to the proper position based on the scroll bar's value.
  if ( bUpdateWindowNow )
    SynchJournalTextWithScrollbar();
}
```

Now that the journal scroll bar is set up correctly, and its control values change whenever new text is added to the journal, we need to allow the user to scroll the text within the journal using that scroll bar. This means that any mouse clicks on the scroll bar need to be detected and dealt with appropriately. So, first things first. How do we detect a click on the Desert Trek game window's scroll bar. When Desert Trek gets a mouse down event in the content region of the main game window, the following function in **Trek Window.c** gets called (we have seen this function before, but we ignored the function to determine if the scroll bar was clicked).

```
void TrekWindowMouseDown( Point pt )
{
  // Convert the where field of the event record from global coordinates to
  // local coordinates.  First, though, make sure that the main game window
  // is the current port, since that's the port the GlobalToLocal() toolbox
  // routine will use when converting from global coordinates.
  SetPort( pWindowTrekWindow );
  GlobalToLocal( &pt );

  // Check to see if the player clicked on the scroll bar.  If not, check to
  // see if the player clicked on any of the picture buttons.
  if ( !ClickedOnJournalScrollbar( pWindowTrekWindow, pt ) )
    ClickedOnCommandsButtons( pt );
}
```

The Desert Trek function, `ClickedOnJournalScrollbar()`, checks to see if the player clicked on the scroll bar and if so, calls a routine to process that click. This function can be found in **Journal.c**.

```
Boolean ClickedOnJournalScroll bar( WindowPtr pWindow,
                                    Point     pt )
{
  Boolean        bClickedOnScrollbar = false;
  ControlHandle  hControl;
  short          sControlPart;

  // Check to see if the player clicked on the scroll bar.
  sControlPart = FindControl( pt, pWindow, &hControl );

  // If they did, handle the click, passing the scroll bar clicked handling
  // routine the part of the scrollbar clicked on by the user.
```

```
  if ( hControl == hControlJournalScrollbar )
  {
    bClickedOnScrollbar = true;
    HandleClickOnJournalScrollbar( sControlPart, pt );
  }

  // Returns true if the user did click on the scroll bar, false if not.
  return( bClickedOnScrollbar );
}
```

The routine that processes the click on the journal scroll bar can also be found in **Journal.c**. This routine needs to track the player's click in the scroll bar, and uses the `TrackControl()` toolbox routine to do so. Remember that the `TrackControl()` routine can take a callback function, one that gets called periodically while the mouse button is down. You would want to supply this callback function in cases where the text should be scrolled while the mouse button is down. When should the text be scrolled while the mouse button is down? The answer is whenever the mouse button was clicked in the up arrow, down arrow, page up region, or page down region. In fact, the only part of the scroll bar where you don't want to specify a callback function is when the user clicks in the thumb part of the scroll bar. Why? When the thumb of the scroll bar, they are allowed to randomly move it to any position, and at any rate. Typically, most Macintosh applications don't scroll the text to track this condition since the text could move anywhere at anytime. On the other hand, when the user clicks in the up or down arrows, or the page up or down regions, the text can be scrolled one line, or one page at a time while the mouse button is held down. This scrolling always occurs in one direction only because the user can't change the scroll direction while the mouse button is down.

```
static void HandleClickOnJournalScrollbar( short  sControlPart,
                                           Point  pt )
{
  // If the user clicked in the thumb of the scroll bar...
  if ( sControlPart == inThumb )
  {
    // Track the click without specifying a callback routine.
    TrackControl( hControlJournalScrollbar, pt, nil );
```

```
    // After the mouse is released, scroll the journal text to the new
    // position.
    SynchJournalTextWithScrollbar();
  }

  // Otherwise, the user clicked on the up/down arrow, or the page up/down
  // region.  Track the click, supplying a callback routine to scroll the
  // the text while the mouse button is down.
  else
    TrackControl( hControlJournalScrollbar, pt, ScrollJournalText );
}
```

The callback function supplied to `TrackControl()` will need to change the current control value of the scroll bar, and scroll the actual text accordingly. Remember, the callback function will be supplied with the control handle of the scroll bar and the part in which the user clicked. The callback function uses the part parameter to determine in what direction to scroll, and by how much. The scroll bar's control value needs to decrease if the user clicked in the up arrow or page up region, and increase if the user clicked in the down arrow or page down region. This is due to the fact that when the scroll bar's control value is 0, the thumb is drawn nearest the up arrow, and when the scroll bar's control value is equal to the maximum control value, the thumb is drawn nearest the down arrow. The following function can be found in **Journal.c**.

```
static pascal void ScrollJournalText( ControlHandle  hControl,
                                      short          sControlPart )
{
  short  sScrollAmount = 0;

  // Depending on the part of the scroll bar in which the player clicked, we
  // need to scroll different amounts.  Either one line at a time or one page
  // at a time, and either up or down.
  switch( sControlPart )
  {
    case inUpButton:

      sScrollAmount = -1;
      break;

case inDownButton:
```

```
            sScrollAmount = 1;
            break;

        case inPageUp:

            sScrollAmount = -sLinesPerPage;
            break;

        case inPageDown:

            sScrollAmount = sLinesPerPage;
            break;
    }

    // If we need to scroll the text...
    if ( sScrollAmount )
    {
        // First set the control value of the scroll bar control to reflect the
        // change.  This will cause the scroll bar control to be drawn correctly
        // (in other words, the thumb part will move to the proper position).
        SetCtlValue(  hControlJournalScrollbar,
                      GetCtlValue( hControlJournalScrollbar ) +
                      sScrollAmount );

        // Scroll the actual text to the correct position.  Chapter 8 on textedit
        // will show this function in detail.
        SynchJournalTextWithScrollbar();
    }
}
```

This concludes the section on controls. Let's now look at dialog boxes and the special way in which they allow you to use controls easily.

Dialog Boxes

A dialog box is a special type of window intended to obtain information from or display information to the user. Dialog boxes share many of the same characteristics of windows, and provide an especially easy way to use controls. In addition to supporting the standard controls described above, they support a mechanism for allowing the user to enter text. Because your game will most likely require that the player enter text at some

point, even if it's just for the high scores list, you will almost certainly use dialog boxes in your game.

Types of Dialog Boxes

The Macintosh operating system provides several types of dialog boxes that can be used by your game; these are *modal dialog boxes*, *modeless dialog boxes*, and *alerts*. Modal dialog boxes require that the user enter the information requested or read the information presented before continuing with your game. In other words, they are modal, which pretty much means that the user has entered a particular mode of the game from which they must exit before they can do anything else. If the user tries to click anywhere outside a modal dialog box, the Macintosh beeps (or flashes the title bar) and ignores that click. An example of when you'd want to use a modal dialog box is when your game prompts the user to enter their name for the high scores list. While this prompt is on the screen, you don't want the player to start another game, clear the high scores list, or do just about anything else. So, you use a modal dialog box and force the player to enter their name before doing anything else (or, if your game supports it, and it should, cancel the dialog box without having to enter their name).

The user can interact with modeless dialog boxes at his or her leisure. The user is not required to enter the requested information and dismiss the dialog box before going on to do other things. Desert Trek uses a modeless dialog box for the high scores window. When the player chooses to see the high scores list, a modeless dialog box appears on the screen, but that dialog box does not have to be dismissed in order for the player to continue playing Desert Trek.

Alerts are a special type of modal dialog box, typically used to display information to the user. They are also used to allow the user to make a choice from several options. An alert generally contains only text and one or more push buttons (okay, and maybe an icon). When the player tries to quit Desert Trek and their current game is not saved, a message is displayed asking them if they want to save before quitting, not save before quitting, or cancel the operation (in other words, not quit). An alert

would be perfect for this scenario because Desert Trek needs to display a message and get a simple response from the user. However, Desert Trek doesn't use alerts for a good reason. I'll explain.

Application Modal Dialog Boxes

Modal dialog boxes were just fine to use back in the good old days when your application was the only thing running on the Macintosh. Remember, modal means that the user must deal with the dialog box before doing anything else. It's that "anything else" part that really becomes a problem when your game runs in a multiprogram environment. It's okay for you to require that the player deal with your modal dialog box before they can do anything else with your game, but you don't want to prevent the player from doing anything at all on their computer before dismissing your modal dialog box. In addition, you probably don't want to prevent the user from seeing your other game windows when a dialog box is displayed on the screen. That box might overlap one of your game windows, and the player might need to see that game window before entering information into that dialog box. If you use a modal dialog box, you would first have to cancel the dialog box to see the window, then bring it up again to enter the requested information. When would this happen? How about a game that asks the user to enter how many troops they'd like to use to attack, but the window showing how many troops they have is being obscured by the dialog box asking the question.

So, why not just always use modeless dialog boxes. Modeless dialog boxes don't lock up the system and can be moved around the screen. It sounds like a decent solution. Unfortunately, doing so would give the user too much freedom. Remember, the reason you used a modal dialog box in the first place was to prevent the user from doing anything else with your game until they dismissed that dialog box. A modeless dialog box would allow the user to start doing things you wouldn't want them doing until they completed the dialog box being displayed.

So, what can you do? What we'd really like is a movable dialog box that's modal to our game, but not to the whole system. Other operating environments, such as Microsoft Windows, support what's called an

application modal dialog box. An application modal dialog box prevents the user from doing anything else with the application displaying the dialog box, but allows the user to click on and use other application's windows. These operating environments also support system modal dialog boxes which prevent the user from doing anything on the system until the dialog box is dismissed. Macintosh's modal dialog boxes and alerts can be considered system modal dialog boxes. System modal dialog boxes should only be used in extreme cases where the user must deal with a situation immediately, such as situations where the entire computer would be affected. Unfortunately, the Macintosh does not support application modal dialog boxes natively. However, they can certainly be simulated with a little work.

All of Desert Trek's modal dialog boxes are application modal dialog boxes (with the exception of the Macintosh's standard File Save and Open dialog boxes which will be discussed in Chapter 9 on Reading and Writing Files). This is accomplished by using modeless dialog boxes, but preventing the user from activating any other Desert Trek window while that modeless dialog box is displayed. Even the alerts that Desert Trek displays are application modal. After I describe how to use dialog boxes in your game, I will show you how to easily implement application modal dialog boxes and alerts.

Dialog Box Records and Pointers

As I briefly stated at the beginning of this section, dialog boxes are a special type of window. In fact, anything that you can do with a window, you can do with a dialog box. This means that you can specify a dialog box pointer in any toolbox call that takes a window pointer. For example, you can set a dialog box's title text or reference constant just as you can with a window. How can this be? It's simple once you look at the dialog box record containing the information for a dialog box. Note that once again, I'm not showing the whole record since you'll use toolbox routines to manipulate the dialog box record elements.

```
typedef WindowPtr DialogPtr;
struct DialogRecord {
     WindowRecord window;
     .
     .
     .
};
```

As you can see, the first element of a dialog box record is a window record. This means that a dialog box is really a window with a few extra things. Also, the `DialogPtr` is defined to be a `WindowPtr`, which means that you don't even need to typecast dialog box pointers when using them in toolbox calls that take window pointers. How convenient.

So, what does a dialog box have that windows do not. In essence, there are two things. First, a dialog box contains a handle to a list of dialog box items. Dialog box items are really just a collection of controls that the dialog box contains. However, the controls are indexed with an item number instead of their control handle. This means that you can identify controls within dialog boxes by their item number, instead of having to keep track of their control handles. The item numbers are defined when you create the dialog box and its corresponding dialog box item table ('DITL' resource) in ResEdit (see Chapter 3). Second, dialog boxes allow you to define any number of text edit boxes, which allows the user to enter text. From your program's point of view, these text edit boxes can be referred to by their item numbers, just like the other standard controls within that dialog box.

Item Types

An advantage of using dialog boxes over windows is that the controls contained within a dialog box can be easily accessed via an item ID. These items include not only standard controls such as push buttons, check boxes, and radio buttons, but also user items, static text fields, text edit fields, icons, and pictures. You determine an item's *type* when defining that item in ResEdit. An item's type is a short integer with one of the following values as defined by the toolbox.

```
enum {
     ctrlItem = 4,          // Standard control.
          btnCtrl = 0,      // Push button.
          chkCtrl = 1,      // Check box.
          radCtrl = 2,      // Radio button.
          resCtrl = 3,      // Control resource.
     statText = 8,          // Static text field.
     editText = 16,         // Text edit field.
     iconItem = 32,         // Icon resource.
     picItem = 64,          // Picture resource.
     userItem = 0           // User item.
};
```

A standard control's item type is the sum of `ctrlItem` and the constant defined for that particular type of control. For example, a radio button's item type is 4 + 2, or 6. Typically, you will only need to worry about item types when you create them with ResEdit.

Static Text and Text Edit Dialog Box Items

As just stated, dialog boxes support editable text fields. In addition, they support *static* text fields. Static text fields are just that, static. The user is not allowed to change the text contained within them. However, your game can certainly change the text displayed within a static text field, as we'll see later. Static text items are typically used as a label for text edit fields, and describe to the user what they are expected to type. Static text fields are also used to convey messages to a user, such as an alert or error condition (e.g., "The name you entered for the high scores list is too long.").

Text edit boxes contained within dialog boxes are an easy way to get text typed from the user. Generally, whenever your game needs to get typed input from the user, such as a name, you will do so through a dialog box containing a text edit field. Text edit fields take care of just about everything for you, (e.g., allowing the user to type characters, change the insertion point, and select text). All you have to do is read the text. We'll learn all about that in a moment.

Loading and Closing Dialog Boxes

To load a dialog box from the resource fork (type **'DLOG'**), use the following toolbox routine.

```
// Loads a dialog from the resource fork.
DialogPtr GetNewDialog(  short      sDLOGResourceID,
                         Ptr        pStorage,
                         WindowPtr  pWindowBehind );
```

Just as with the `GetNewWindow()` toolbox routine, to load a window from the resource fork, you need to specify the resource ID of the dialog box to be loaded. The dialog box item's table of controls will also automatically be loaded from the 'DITL' resource with the same ID. You are also given the opportunity to allocate your own storage for the dialog box record by specifying a pointer to that storage in the `pStorage` parameter. Passing `nil` will cause the Macintosh to automatically allocate and maintain the storage for you. This is the recommended route. Last, you need to specify the window which the loaded dialog box will appear behind. Just like the `GetNewWindow()` toolbox routine, -1 will cause the dialog box to appear in front of all windows, `nil` will cause the dialog box to appear behind all window.

When you are finished with a dialog box (typically when the user dismisses the dialog box), you need to remove it from the screen as well as release the memory used by its dialog box record. There are two toolbox routines that close a dialog box. The one you will need depends on whether or not you choose to allocate storage for the dialog box record yourself.

```
// Closes a dialog, removing it from the screen.  The memory
// used by the dialog record is not automatically freed, so
// you will need to do so yourself.  Use this call if you
// specified a pointer for pStorage in the GetNewDialog()
// call.
void CloseDialog( DialogPtr pDialog );
```

```
// Closes a dialog, removing from the screen.  The memory
// used by the dialog record is automatically freed.  Use
// this call if you specified nil for pStorage in the
// GetNewDialog() call.
void DisposDialog( DialogPtr pDialog );
```

Examples of these calls will be given shortly.

Accessing Dialog Box Items

Remember that one of the advantages of dialog boxes is that they contain a list of dialog box items that can be easily manipulated. These items can be accessed by using either the item's ID or the item's handle. Some of the toolbox routines used to manipulate dialog box items require the item's ID, and some require the item's handle. When you create a dialog box and its item list with ResEdit, you assign IDs to all the items. What you need is a way to obtain a dialog box item's handle from its ID, which you already know. The following toolbox routine is exactly what you need.

```
// Get a dialog item's type, handle, and rectangle given it's
// ID.
void GetDItem(  DialogPtr  pDialog,
                short      sItemID,
                short      *psItemType,
                Handle     *phItem,
                Rect       *pRectItem );
```

This toolbox routine returns a dialog box item's type, handle, and rectangle. For the most part, you will need only the handle returned. Because you defined the item's type when you created it with ResEdit, you probably won't need that value, unless you expect an item's type to change during program execution (which could easily be confusing to the user). Occasionally, you'll need an item's rectangle, especially if you are manually drawing anything in the dialog box related to the item in question. In the next chapter on Quickdraw, we'll see an example of how to use a user type item to draw a pop-up menu in a dialog box. In that example, we'll need the item's rectangle in order to draw the shadowed rectangle showing the user where to click to pop-up the menu.

You can set an item's information by using the following toolbox call.

```
// Set a dialog item's type, handle, and rectangle given it's
// ID.
void SetDItem(  DialogPtr  pDialog,
                short      sItemID,
                short      sItemType,
                Handle     hItem,
                Rect       rectItem );
```

Rarely, if ever, will you need to use this call. The only time you'll need to use it is when you want to change a dialog box item's location. After moving a control using the `MoveControl()` routine to position an item in a dialog box, you need to use `SetDItem()` to have the change take place. You need to use both toolbox calls, otherwise the item's new position won't get reflected onscreen. The following function from Desert Trek moves the push buttons contained in the high scores dialog box. The function can be found in **Scores Window.c**.

```
#define BUTTON_SPACING          90

static void MoveScoresButtons( void )
{
  Rect    rectItem;
  short   sItemType;
  short   sItemWidth;
  short   sItemHeight;
  Handle  hItem;
  short   sLoop;
  short   sDistance = BUTTON_SPACING;

  // Loop through all three buttons in the high scores dialog.
  for( sLoop = PB_OK; sLoop <= PB_CLEAR_ALL; sLoop++ )
  {
    // Get the button's item type, handle, and rectangle.
    GetDItem( pDialogScores, sLoop, &sItemType, &hItem, &rectItem );

    // Determine where the button should go.  The first button goes in the
    // lower right corner of the dialog, and the others fan out to the left
    // of it.
    sItemWidth = rectItem.right - rectItem.left;
    sItemHeight = rectItem.bottom - rectItem.top;
```

```
        SetRect( &rectItem, rectScores.right - sDistance, rectScores.bottom + 4,
                rectScores.right - sDistance + sItemWidth,
                rectScores.bottom + 4 + sItemHeight );

        // Move the button control to it's new location.
        MoveControl(  (ControlHandle) hItem, rectItem.left,  rectItem.top );

        // Set the button control's rect in the dialog item list.
        SetDItem( pDialogScores, sLoop, sItemType, hItem, &rectItem );

        // Prepare for the next button's position.
        sDistance += BUTTON_SPACING;
    }
}
```

Getting and Setting Text for Static Text Items and Text Edit Fields

The toolbox provides several routines for retrieving and setting the text of static text and text edit items within a dialog box. Two of these routines take the item's handle, which you obtain by using `GetDItem()`.

```
// Sets the text of a static text or text edit field of a
// dialog.
void SetIText(  Handle  hItem,
                Str255  str255Text );

// Gets the text of a static text or text edit field of a
// dialog.
void GetIText(  Handle  hItem,
                Str255  str255Text );

// Sets the selection range of a text edit field.  The
// start and end positions are character positions.
void SelIText(  DialogPtr  pDialog,
                short      sItemID,
                short      sSelectionStart,
                short      sSelectionEnd );
```

In addition to selecting the text in the specified text edit field, `SelIText()` also sets the input focus to the specified field. The input

focus determines which field will get keystrokes typed by the user. Generally, it's the field with the blinking insertion point. There are a couple of cases in which you may want to use `SelIText()`. The first is when you load any dialog box that contains one or more text edit fields. You should select all the text contained within the field you want to initially have focus. The second is when you are checking input fields for errors. If a field contains an error, you should set focus to that field and select all its text after displaying the error message. This is especially useful to the user in dialog boxes with multiple text edit fields because, if you select that field, the user will immediately know where the error lies. We'll see an example of this later on.

Parameterized Text

When you define a dialog box item table in RedEdit (resource type **DITL**), you can place parameter tokens into the text of any item defined (static text items, radio buttons, push buttons, etc.). The parameter tokens are ^0, ^1, ^2, and ^3, meaning that you can have up to four unique parameter tokens per dialog box. The same token can be used more than once in a dialog box if you want the same parameterized text to appear in more than one dialog box item. In your code, you can set the text to be substituted for these tokens by using the following toolbox call.

```
// Sets the 4 parameterized text tokens, which will get
// substituted into your alerts and dialogs.  This is a
// global change, meaning that unless you reset them, they
// will be used in all dialogs and alerts subsequently
// displayed by your program.
void ParamText(  Str255     str255Token0,     // "^0"
                 Str255     str255Token1,     // "^1"
                 Str255     str255Token2,     // "^2"
                 Str255     str255Token3 );   // "^3"
```

Just remember that these substitutions will be made in any dialog box or alert loaded by your program. This means that you should probably set them just before loading any dialog box or alert that needs them.

Showing and Hiding Dialog Box Items

Occasionally, you may need to show or hide dialog box items. The following toolbox routines allow you to do so. Remember that it is better practice to disable an item rather than make it invisible.

```
// Hides a dialog item, making it invisible.
void HideDItem(  DialogPtr  pDialog,
                 short      sItemID );

// Shows a dialog item, making it visible.
void ShowDItem(  DialogPtr  pDialog,
                 short      sItemID );
```

Finding an Item Based on the Mouse Location

Normally, dialog boxes take care of all mouse clicks on dialog box items for you. The dialog box automatically tracks the mouse click and returns to your program the item ID of the dialog box item clicked on by the user. However, if for some reason you need to determine which item the mouse is over, the toolbox provides a routine to tell you which dialog box item is located at a given point.

```
// Given a point in local coordinates, tells you which dialog
// item is under that point.
short FindDItem(  DialogPtr  pDialog,
                  Point      pt );
```

Drawing Dialog Boxes

The following routines are used to draw a dialog box and its contents (including all dialog box items).

```
// Draws an entire dialog and it's contents.
void DrawDialog( DialogPtr pDialog );

// Draws the specified region of dialog and it's contents.
void UpdtDialog(  DialogPtr  pDialog,
                  RgnHandle  hRegion );
```

The two toolbox routines, `ModalDialog()` and `DialogSelect()`, which will be discussed shortly, automatically draw a dialog box's contents when an update event occurs for that dialog box. This means that if you don't do anything unusual, such as custom drawing within a dialog box, you won't need to use the `DrawDialog()` or `UpdtDialog()` calls. However, Desert Trek uses these calls since all dialog box update events are processed to allow for custom drawing in any dialog box. So, we'll see `DrawDialog()` used in the dialog box examples shown later in this chapter.

Using Alerts

An alert is a special type of modal dialog box that displays a message and asks for a simple response from the user. An example would be asking the user a yes/no question such as asking whether or not the user wants to save a game before quitting. The toolbox calls to display alerts take an alert resource ID as an input parameter. You specify this ID when creating the alert with ResEdit. These routines return the item ID clicked by the user to dismiss the alert. This is how you determine whether the user clicked **OK**, **Cancel**, **Yes**, **No**, etc. The alert toolbox calls also take as a parameter a pointer to what's called a *filter procedure*. Filter procedures will be discussed shortly in the section on getting modal dialog box events. The only difference between the various alert calls is the icon displayed in the alert. See Figure 6.3 for the icons displayed by the toolbox calls described below (`Alert()` does not automatically display an icon; you need to define it yourself when creating the alert in ResEdit).

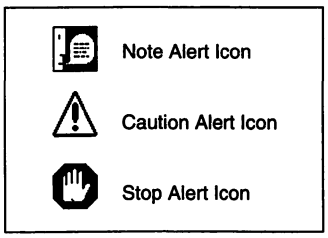


Figure 6.3 The alert icons.

```
// Displays a generic alert.
short Alert(  short    sAlertID,
              ProcPtr  pProcFilter );

// Displays a note alert.
short NoteAlert(  short    sAlertID,
                  ProcPtr  pProcFilter );

// Displays a caution alert.
short CautionAlert(  short    sAlertID,
                     ProcPtr  pProcFilter );

// Displays a stop alert.
short StopAlert(  short    sAlertID,
                  ProcPtr  pProcFilter );
```

Again, Desert Trek does not use these toolbox routines because they prevent the user from doing anything else on the computer until the alert is dismissed. Instead, Desert Trek simulates application modal alerts by using modeless dialog box boxes, which do not prevent the user from switching to another application. You can very easily use this code, which will be shown shortly, in your own game.

Using Modal Dialog Boxes

When your game displays a modal dialog box on the screen, you need a way to cause that dialog box to take control of the system and report back to your program any clicks on items within that dialog box. The

Macintosh toolbox provides a routine to make it very easy to accomplish this task. Before using this call, make sure the dialog box is loaded and the frontmost window (notice that this routine does not take a dialog box pointer as a parameter, so the dialog box in question must be the frontmost window in order for this call to work).

```
// Gives control of the Macintosh over to a dialog.  Any user
// interaction with the dialog is automatically handled.
// This routine returns when the user clicks on a dialog
// item.  Keep calling this routine in a while() loop until
// the user clicks on an item that dismisses the dialog.
void ModalDialog(  ProcPtr  pProcFilter,
                   short    *psItemHit );
```

This toolbox routine takes two parameters: a *pointer to a filter function*, and a *pointer to a short integer*, which will get set to the item ID of the item clicked on by the user. This toolbox call does not return control to your program until the user actually selects a dialog box item. In other words, if you click on a button, but change your mind and move the mouse out of that button before releasing the button, `ModalDialog()` will not return control to your program. If you specify `nil` for the filter function parameter, a standard filter function will be used. This standard filter function does one thing: converts the Return key or Enter key keystroke to a click on the item with an ID of 1. Thus, the item with an ID of one is typically referred to as the *default* item of a dialog box. Usually, it should be your OK button, or its equivalent. If you need to do anything special in the modal dialog box, such as custom drawing or converting other keystrokes to actions, you must supply your own filter function. If you do so, you must take over the task of converting Return and Enter key keystrokes to clicks on item 1. The filter function is in fact a callback routine that the Macintosh calls whenever an event occurs to the modal dialog box used by your game. A filter function must be declared as follows:

```
// Filter function definition for ModalDialog() and
// the ShowAlert() family of toolbox routines.
pascal Boolean MyFilterProc(  DialogPtr    pDialog,
                              EventRecord  *pEvent,
                              short        *psItemHit );
```

Your filter function will receive the dialog box pointer for which the event occurred (this will always be the modal dialog box currently displayed), and the event record for the event itself, which can be modified if needed. If your filter function processing determines that this event should cause an item to be clicked, you need to set that item number and return it using the `psItemHit` parameter. For example, if you are mapping the Escape key to a Cancel button, you should set `psItemHit` to the item ID of the Cancel button if the event is a key down event of the Escape key. Lastly, the filter function returns `true` or `false`, depending on if you want `ModalDialog()` to return the item ID to the portion of your code that called `ModalDialog()`. Returning true will cause your program to regain control at the `ModalDialog()` call. You should return true if you set the `psItemHit` parameter.

Though Desert Trek doesn't use any modal dialog boxes, a typical modal dialog box routine would look something like the following code example. The use of a filter function is not shown. If you really need to use a filter function, you should seriously consider using a modeless dialog box instead (or, the application modal dialog box Desert Trek simulates).

```
static void DoModalDialog( void )
{
  DialogPtr pDialog;
  Boolean   bExit = FALSE;
  short     sItemHit;

  // Load the modal dialog from the resource fork.
  pDialog = GetNewDialog( DLG_MODAL,  nil, (WindowPtr) -1 );

  // Initialize any dialog items here.  Things you might do include setting
  // the text of any text edit fields, and selecting any radio buttons or
  // check boxes that need to be set.

  // Show and select the dialog.  It must be the frontmost window before
  // calling ModalDialog().
  ShowWindow( pDialog );
  SelectWindow( pDialog );

  // Loop until the user selected something in the dialog that would cause it
  // to go away.
```

```
  while( !bExit )
  {
    // Get the next dialog item selected by the user.  No filter function is
    // specified.
    ModalDialog( nil, &sItemHit );

    // Take the appropriate action, depending on which item the user clicked.
    switch( sItemHit )
    {
      // The user clicked on the OK button.  Usually, you would read the
      // states of any dialog items the user was allowed to change.  This
      // includes reading text edit fields and the state of any check boxes
      // and radio buttons.  Also, since the user now wishes to dismiss the
      // dialog, set the flag to exit the while loop.
      case OK:

        bExit = TRUE;
        break;
    }
  }

  // The user has dismissed the dialog, so remove it from the screen and free
  // it from memory.
  DisposDialog( pDialog );
}
```

Using Modeless Dialog Boxes

Modeless dialog boxes are easy to use once you know what to do. In fact, they are probably just as easy to use as modal dialog boxes that require a filter function, and they are much more versatile. For the most part, you can treat modeless dialog boxes like any standard window. The advantage of modeless dialog boxes, though, is that they support dialog box items such as text edit boxes, and they report clicks on any of their items to your game in a manner that's easy to process. You don't need to use `FindControl()` and `TrackControl()`, which are required when processing click events for control items in standard windows.

There are a couple of things that you need to do differently from standard windows. In order to get the convenient dialog box item processing, you have to use a special toolbox call that processes modeless dialog box events. This question arises: How do I know when an event is destined for

a modeless dialog box or a standard window? The Macintosh toolbox provides a call to tell you if an event is meant for a modeless dialog box.

```
// Returns true if the given event is a dialog event.
Boolean IsDialogEvent( EventRecord *pEvent );
```

If the event is a dialog box event, you need to call the following toolbox routine to determine what dialog box item, if any, was clicked on by the user.

```
// Handles a dialog event.  Returns true if you need to
// respond to the message (in other words, an item within
// the dialog was clicked on by the user).  ppDialog will
// contain the dialog pointer of the dialog effected by the
// event.  psItemHit will be set to the item id within that
// dialog clicked on by the user. You only supply pEvent,
// the other parameters are really return values.
Boolean DialogSelect(  EventRecord  *pEvent,
                       DialogPtr    *ppDialog,
                       short        *psItemHit );
```

If `DialogSelect()` returns `true`, the user clicked on one of the dialog box items. The ID of that item is returned in `psItemHit`. If `DialogSelect()` returns `false`, the user did not click on one of the items. You may decide to take further action based on the event type. For example, if you allow the user to click on objects within the dialog box that aren't in the dialog box's item table, you need to determine if the event was a mouse down event in one of those objects. Desert Trek needs to do this in the high scores window. The skill level indicators at the top of the dialog box are not items, so any user clicks in the dialog box not detected by `DialogSelect()` need to be examined to see of they were located within any of the skill level indicators.

In Chapter 2 in the section on events, it was stated that even if `WaitNextEvent()` or `GetNextEvent()` returns `false`, meaning that no event was posted to your game, you need to check to see if a dialog box event was posted anyway. Why? Well, modeless dialog boxes with text edit boxes occasionally need to blink the cursor within the active text edit box. This is accomplished when you call `DialogSelect()`. In order to force your program to periodically call `DialogSelect()`,

`WaitNextEvent()` and `GetNextEvent()` occasionally return a null dialog box event meant simply to cause the cursor in a modeless dialog box to blink. Go back and reread the code example in Chapter 2 showing Desert Trek's `CheckEvent()` function to see this in action.

Modeless Dialog Box Example

Let's look at all the code related to one of Desert Trek's modeless dialog boxes. The high scores dialog box is a modeless dialog box, meaning that it coexists with all of Desert Trek's other windows and modeless dialog boxes. The player is allowed to show the High Scores dialog box and keep it around while they play Desert Trek. Let's first look at the code to create the high scores dialog box. Notice that I make several calls to window routines, passing in the dialog box pointer as a parameter. Remember, dialog box pointers are synonymous with window pointers.

```
static DialogPtr pDialogScores;
static short     sSkillLevelViewed;
static Rect      rectScores;

void ConstructScoresWindow( void )
{
  // If the high scores dialog is already loaded, just make it the
  // frontmost window.
  if ( pDialogScores )
    SelectWindow( pDialogScores );

  // Otherwise, we need to load the dialog and show it on the screen.
  else
  {
    // The skill level initially shown in the high scores dialog is the
    // skill level currently selected by the user.
    sSkillLevelViewed = GetSkillLevel();

    // Create the offscreen components of the high scores window.  All
    // drawing first takes place offscreen before being displayed in the
    // dialog.  Chapter 7 on quickdraw will cover this practice thoroughly.
    ConstructScoresWindowOffscreen();

    // Load the dialog from the resource fork.
```

```
        pDialogScores = GetNewDialog( DLG_SCORES,  nil, (WindowPtr) -1 );

        // Set the dialog's reference constant so that the dialog can be
        // identified later by it's reference constant.
        SetWRefCon( pDialogScores, SCORES_WINDOW_ID );

        // Size the dialog to the correct dimensions.
        SizeWindow( pDialogScores, rectScores.right, rectScores.bottom + 26,
                    false );

        // Move the buttons in the dialog's item list to their proper locations.
        // We've already seen the code for this routine above.
        MoveScoresButtons();

        // Draw the high scores offscreen.  Again, the chapter on quickdraw will
        // cover the details.
        DrawScoresWindow();

        // Call the generic Desert Trek routine to center a window on the screen.
        CenterWindow( &pDialogScores->portRect, nil, true, pDialogScores );

        // Make the dialog visible and the frontmost window.
        ShowWindow( pDialogScores );
        SelectWindow( pDialogScores );
    }
}
```

Once the High Scores dialog box is loaded, we need to process events for that dialog box. Most of the window-related events, such as clicks in the title bar or close box, are handled in the `HandleMouseEvent()` routine in **Main.c**. This routine does not distinguish between specific windows or dialog boxes for mouse clicks in the title bar or close box regions. For mouse clicks in the content region of a dialog box or window, though, the High Scores dialog box is singled out by using its reference constant. If a mouse click in the content region of the High Scores window is detected, `DoScoresWindowEvent()` is called. In addition, keystrokes made by the user while the High Scores dialog box is the active window are sent to `DoScoresWindowEvent()`. The `DoScoresWindowEvent()` routine shown below can be found in **Scores Window.c**. Notice the use of `DialogSelect()` before doing any other processing.

```
void  DoScoresWindowEvent( EventRecord  *pEvent )
{
  DialogPtr  pDialog;
  short      sItem;
  short      sLoop;

  // Since the scores window is a dialog, call DialogSelect to see if the
  // event was a user click in one of the dialog items.  If so, sItem will
  // contain the ID of the item clicked.
  if ( DialogSelect( pEvent, &pDialog, &sItem ) )
    switch( sItem )
    {
      case PB_OK:

        // The OK button dismisses the high scores window.
        DestructScoresWindow();
        break;

case PB_CLEAR:

        // Clear the high scores for the skill level being viewed.
        ClearScores( sSkillLevelViewed - 1 );

        // Call the ViewNewLevel() routine which will force a redraw of the
        // high scores window (now with the cleared scores).
        ViewNewLevel( sSkillLevelViewed );
        break;

case PB_CLEAR_ALL:

        // Clear the high scores for all skill levels.
        for( sLoop = 0; sLoop < 10; sLoop++ )
          ClearScores( sLoop );

        // Reflect the changes by redrawing the high score window.
        ViewNewLevel( sSkillLevelViewed );
        break;
    }

  // If the event posted was not an event affecting one of the dialog items,
  // and the event was a mouse down event, check to see if the player clicked
  // on one of the skill level indicators.
  else if ( pEvent->what == mouseDown )
    ScoresWindowMouseDown( pEvent->where );
```

```
  // If the event was a keystroke, process it.  The ScoresWindowKeyDown()
  // routine takes the character code of the keystroke as a parameter.
  else if ( pEvent->what == keyDown )
    ScoresWindowKeyDown( BitAnd( pEvent->message, charCodeMask ) );
}
```

The `ScoresWindowMouseDown()` and `ScoresWindowKeyDown()` routines will not be shown here, but feel free to look them up in the **Scores Window.c**. The last event type we need to worry about for the High Scores dialog box is the update event. Note that Desert Trek does not call `DialogSelect()` in response to an update event destined for the High Scores window. Since the High Scores window has many items that need to be custom drawn, an `UpdateScoresWindow()` routine was created to perform all of the dialog box's drawing, including that which would occur if `DialogSelect()` was called. This routine can also be found in **Scores Window.c**. Most of the routine contains drawing functions that will be covered in Chapter 7 on QuickDraw so don't worry too much about them right now.

```
void UpdateScoresWindow( void )
{
  GrafPtr   pGrafCurrent;
  RGBColor  rgbForeColorCurrent;
  RGBColor  rgbBackColorCurrent;
  hColors   hStd16Colors = GetColorsHandle();
  Boolean   bUsingColorGraphics;

  bUsingColorGraphics = UsingColorGraphics( nil );

  // If the dialog pointer is valid, let's draw the dialog.
  if ( pDialogScores )
  {
    GetPort( &pGrafCurrent );
    SetPort( pDialogScores );

    // Start the update process, which will automatically clip any drawing
    // outside the region the needs redrawing.
    BeginUpdate( pDialogScores );

    // Call the toolbox routine that automatically draws the dialog's items.
    // We need to do this before doing any custom drawing since if we called
    // this after the custom drawing, the dialog items may overwrite the
```

```
    // custom drawing we did.
    DrawDialog( pDialogScores );

    // Perform the custom drawing, which pretty much just copies the offscreen
    // high scores area to the onscreen dialog.
    GetForeColor( &rgbForeColorCurrent );
    GetBackColor( &rgbBackColorCurrent );
    RGBForeColor( &(*hStd16Colors)->rgbColor[ColorBlack] );
    RGBBackColor( &(*hStd16Colors)->rgbColor[ColorWhite] );

    CopyBits( &bitmapScores, &pDialogScores->portBits, &rectScores,
              &rectScores, srcCopy, nil);

    RGBForeColor( &rgbForeColorCurrent );
    RGBBackColor( &rgbBackColorCurrent );

    // End the update process.
    EndUpdate( pDialogScores );
    SetPort( pGrafCurrent );
  }
}
```

When the user dismisses the High Scores dialog box, either by pressing the **OK** push button, or clicking on the dialog box's close box, the dialog box needs to be removed from the screen, and the dialog box record freed from memory. The following routine from **Scores Window.c** does just that.

```
void DestructScoresWindow( void )
{
  // Clean up the offscreen components of the high scores dialog.
  DestructScoresWindowOffscreen();

  // If the dialog pointer is valid...
  if ( pDialogScores )
  {
    // Remove the dialog from the screen, and free up the memory taken by
    // its dialog record.
    DisposDialog( pDialogScores );

    // Set the dialog pointer to nil, so that we know the dialog is no longer
    // on the screen.
    pDialogScores = nil;
  }
}
```

That concludes the modeless dialog box example. As you can see, supporting modeless dialog boxes is not difficult at all.

Supporting Application Modal Dialog Boxes

We just saw an example of how to support modeless dialog boxes in your game. However, there are times when you want to show a dialog box and prevent the player from interacting with the other windows and dialog boxes of your game until that dialog box is dismissed. Normally, you would use a modal dialog box, but as was discussed, modal dialog boxes are bad because they prevent the user from accessing the windows of other applications running on the Macintosh.

The trick to supporting Application Modal dialog boxes is to implement modeless dialog boxes while at the same time preventing the user from interacting with any of your other windows and dialog boxes. This task isn't very difficult and only requires one check to be added to your mouse down event processing. Okay, there's also one other thing to think about. How can the user interact with one of your game's dialog boxes or windows? The most obvious is by clicking on it. We've just covered that. However, it's also possible to select and/or affect game windows and dialog boxes by choosing various menu commands. So, you need to disable any menu commands in your game that affects any window or dialog box. To be honest, that's probably just about every menu command your game supports. No, don't go thinking that the Quit command doesn't affect any windows or dialog boxes. It causes all of your windows and dialog boxes to close, and most likely creates a dialog box itself, asking if the user wants to save before quitting. Basically, you need to disable all of your game's menus when an application modal dialog box is displayed. The only menu commands you want to remain enabled are the items under the apple menu not associated with your game, and any system menus such as the Balloon Help menu and Application menu.

So, let's see some code examples of what needs to be done to support application modal dialog boxes. First, let's look at how mouse clicks are handled. The following routine comes from **Main.c**. We've seen it before, but now look at it again, paying special attention to the code that checks to see if an application modal dialog box is currently active.

```
static void HandleMouseEvent( EventRecord *pEvent )
{
  short      sWindowPart;
  short      sWindowID;
  WindowPtr  pWindow = nil;
  Rect       rectDragArea;

  // Get the window pointer, window part, and window ID of the window clicked.
  sWindowPart = FindWindow( pEvent->where, &pWindow );
  sWindowID = (short) GetWRefCon( pWindow );

  // If the click was in a system window (pretty much any window not belonging
  // to this game), let the toolbox call SystemClick() handle it.
  if ( sWindowPart == inSysWindow )
    SystemClick( pEvent, pWindow );

  // If the click occurred in the menu bar, handle the menu selection.
  // Remember, if an application modal dialog is active, all the menus
  // commands related to this game are disabled.  So, we're really checking
  // for system menu clicks in that case (apple, balloon help, or application
  // menus).
  else if ( sWindowPart == inMenuBar )
    HandleMenuSelection( MenuSelect( pEvent->where ) );

  // If an application modal dialog is active, and the window for which this
  // click event occurred was NOT for that application modal dialog, simply
  // beep and ignore the event.  This prevents other windows and dialogs from
  // getting activated while an application modal dialog is up.
  else if ( ( IsAppModalDialogUp() ) &&
            ( pWindow != GetAppModalDialogWindow() ) )
    SysBeep( 1 );

  // Activate the clicked window if it wasn't topmost.
  else if ( pWindow != FrontWindow() )
    SelectWindow( pWindow );

  // Handle the actual click.  We'd get here if no application modal dialog
  // was up, or if the click was on the application modal dialog (in which
  // case, we need to handle it).
  else switch ( sWindowPart )
  {
    case inDrag:

      rectDragArea = screenBits.bounds;
```

```
        DragWindow( pWindow, pEvent->where, &rectDragArea );
        break;

    case inGoAway:

    if ( TrackGoAway( pWindow, pEvent->where ) )
        HandleCloseWindow( sWindowID );
        break;

    case inContent:

        HandleMouseDownInContent( pEvent, sWindowID );
        break;
    }
}
```

As you can see, one `else` clause in the mouse down event handling routine allows us to implement application modal dialog boxes. Well, okay, there are several supporting functions that need to be discussed, but this is the crux of the support for application modal dialog boxes.

Before looking at some of those supporting functions, let's look at the code to disable Desert Trek's menus when an application modal dialog box gets displayed. This routine gets called either when an application modal dialog box is first displayed, or when it is dismissed and there are no other application modal dialog boxes active. In a moment, we'll discuss how more than one of these boxes might be displayed at one time (of course, only one of them can be active).

```
void SetModalDialogMenuState( Boolean bEnable )
{
  // An application modal dialog is active.  Disable all Desert Trek menus.
  // Note that we can disable every menu wholesale except the Apple menu.
  // For the Apple menu, we should only disable Desert Trek's menu items, not
  // the menu folder items/desk accessories.
  if ( !bEnable )
  {
    DisableItem( hMenuApple, AppleMenuAboutID );
    DisableItem( hMenuApple, AppleMenuHelpID );
    DisableItem( hMenuApple, AppleMenuCarysGamesID );
    DisableItem( hMenuFile, 0 );
    DisableItem( hMenuOptions, 0 );
```

```
    DisableItem( hMenuCommands, 0 );
    DisableItem( hMenuBuy, 0 );
    DisableItem( hMenuSkillLevel, 0 );
    DrawMenuBar();
  }

  // The last application modal dialog has been dismissed.  First, enable all
  // of Desert Trek's menus, and then call the AdjustMenus() routine which
  // will set all of the menu's enabled/disabled states based on the current
  // game's conditions.
  else
  {
    EnableItem( hMenuApple, AppleMenuAboutID );
    EnableItem( hMenuApple, AppleMenuHelpID );
    EnableItem( hMenuApple, AppleMenuCarysGamesID );
    EnableItem( hMenuFile, 0 );
    EnableItem( hMenuOptions, 0 );
    EnableItem( hMenuCommands, 0 );
    EnableItem( hMenuBuy, 0 );
    EnableItem( hMenuSkillLevel, 0 );
    AdjustMenus();
  }
}
```

We have now seen how your code can allow modeless dialog boxes to act like application modal dialog boxes, but how do you actually implement the application modal dialog box? In other words, how do you create this dialog box, process events for it, and react when the user dismisses it? Desert Trek has defined a unit, **App Modal Dialog box.c**, which does all of this for you. All you need to do is include this unit in your game and use its functions to support application modal dialog boxes (or, if you really want, you can copy and change the code to suite your specific needs). The only function this unit uses externally is the `CenterWindow()` function found in **Common Functions.c**. You'll need to make sure that you have a `CenterWindow()` function defined somewhere in your game, or just copy this function from **Common Functions.c** to **App Modal Dialog.c**.

Let's look at the external functions defined in **App Modal Dialog.c** and see when you should use them in your game. Probably the first thing you'd want to do is create an application modal dialog box. This is by far

the most complex function in the unit because it does all the setup work. Here's the prototype:

```
DialogPtr ConstructAppModalDialog( short sDialogID,
                                   void  (*pfnEvent)( short,
                                                      DialogPtr,
                                                      EventRecord *),
                                   void  (*pfnDraw)( DialogPtr ),
                                   void  (*pfnHandleMenus)( Boolean ),
                                   short sDefaultAction,
                                   short sEscapeAction,
                                   short sYesAction,
                                   short sNoAction );
```

There are quite a few parameters, but you don't need to specify all of them. They are there to provide plenty of flexibility. The first parameter, `sDialogID`, is used to specify the resource ID of the 'DLOG' resource. The next three parameters are callback functions that you want called when certain things happen to the application modal dialog box. Do not define these callbacks as `pascal`, as you would when supplying callback functions to toolbox calls, since they all need to use the C calling convention.

The first callback function you supply is the event callback function, `pfnEvent`. You will always provide an event callback function because you need to process events for the dialog box (how can the user dismiss the dialog box if you don't process the event for it?). Essentially, the event callback function should look exactly like any function that processes events for a modeless dialog box (see the example of the `DoScoresWindowEvent()` function shown in the previous section). Your event callback function should take three parameters, a short integer giving you the ID of the selected item, a dialog box pointer of the modeless dialog box in question, and a pointer to the event record in case you need to do any special event processing (like supporting pop-up menus).

The second callback function that you can optionally supply is `pfnDraw`, a custom dialog box update routine. You would only need to supply this parameter if you need to do any custom drawing in the dialog

box. Your drawing callback function will be passed to the dialog box pointer. Note that `BeginUpdate()` and `EndUpdate()` are called automatically for you, so your custom drawing routine simply needs to do its drawing. It doesn't have to concern itself with setting the drawing port, calling `BeginUpdate()`, etc.

The third callback function you can optionally supply is a pointer to an application modal menu enabling and disabling routine. This routine takes one parameter telling you whether or not to enable or disable menus (a value of `true` means to enable the menus because no application modal dialog box is left on the screen). Your menu callback routine will be called whenever you need to enable or disable the menus. In other words, when the first application modal dialog box appears on the screen, or when the last one dismissed. You just saw Desert Trek's menu callback routine, `SetModalDialogMenuState()` above.

The last four parameters taken by `ConstructAppModalDialog()` are keyboard shortcuts for dialog box items. Specify the item ID of the item you want clicked when you press the **Return** or **Enter** key in `sDefaultAction`, the **Escape** or **Tilde** (~) key in `sEscapeAction`, the **Y** or **y** key in `sYesAction`, and the **N** or **n** key in `sNoAction`. Specify 0 for those key combinations you don't want to carry any special meaning. For those chosen items, your event handling routine will give the specified item IDs when the user types the corresponding key on the keyboard.

Note that `ConstructAppModalDialog()` does not show the dialog box. This allows you to do some initialization before the dialog box appears on the screen, such as setting text edit fields or radio button and check box states. Make sure to show the dialog box yourself after calling `ConstructAppModalDialog()`.

The remaining functions defined in **App Modal Dialog.c** used to support application modal dialog boxes are really very simple. Use the following routines to determine if an application modal dialog box is on the screen, and to retrieve its dialog box pointer. See the `HandleMouseEvent()` routine shown above to see how these two functions should be used in your code.

```
// Returns true if an application modal dialog is on the screen.
Boolean IsAppModalDialogUp( void );

// Returns the dialog pointer of the application modal dialog on the screen.
// If you receive an event for a window or dialog other than the dialog
// returned by this call, ignore the event (assuming, of course, that the
// above function returned true).
DialogPtr GetAppModalDialogWindow( void );
```

How do you pass events to the active application modal dialog box? In the same way you pass events to any of your other windows or dialog boxes. That's accomplished by checking the reference constant of the dialog box. The following reference constant is set for all application modal dialog boxes:

```
// The reference constant for any application modal dialog.  Use it to know
// when to pass events to UpdateModalDialog() and DoAppModalEvent() described
// below.
#define APP_MODAL_DIALOG_ID    5
```

When you receive an event for a dialog box with a reference constant of APP_MODAL_DIALOG_ID, call one of the following two routines (one is for mouse and keyboard events, the other is for update events).

```
// Call this routine when an application modal dialog gets an update event.
// Pass it the window handle you extracted from the event record.
void  UpdateModalDialog( DialogPtr pDialog );

// Call this routine when an application modal dialog gets a mouse or keyboard
// event.  Pass it the pointer to the event.
void  DoAppModalEvent( EventRecord *pEvent );
```

When your routine that handles application modal dialog box events wants to dismiss the dialog box due to some user action, call the following routine. Again, you need to pass it a pointer to the function you defined to deal with the disabling and enabling of menus in response to application modal dialog boxes.

```
// Removes the application modal dialog from the screen.
void  DestructAppModalDialog( void (*pfnHandleMenus)(Boolean) );
```

One final note about Desert Trek's support for application modal dialog boxes. The **App Modal Dialog.c** unit supports up to five levels of application modal dialog boxes. In other words, up to five application modal dialog boxes can be on the screen at a time. Keep in mind, however, that the user is allowed to interact with only one of them at a time, and they must dismiss the active dialog box before having access to the one underneath. You should never need more than two levels of application modal dialog boxes on the screen at any given time, so five is being very generous. When would you want to have more than one application modal dialog box on the screen at a time? The answer is whenever an application Model dialog box causes another dialog box or alert to be displayed. For example, if you have an application modal dialog box that asks the user to type in their name, and the name typed in is invalid or contains too many characters, you need to display a message stating so. That message will most likely be an application modal dialog box (or alert, as will be described in the next section). This means that there will be two of these boxes on the screen: the original dialog box asking for the player's name, and a message stating that the name they tried to enter is invalid. The player must dismiss the topmost box before returning to the first.

Let's see how Desert Trek uses the application modal dialog box support provided by **App Modal Dialog.c**. The following examples are used to display a dialog box asking the player to enter their name when they achieve a high score. The following function, found in **Scores Window.c**, creates the application modal dialog box when the user achieves a high score.

```
void CheckHighScore( void )
{
  PGAME_STATE  pGameState;
  DialogPtr    pDialog;

  // Get the game's state, so that the player's score can be computed.
  pGameState = RetrieveGameState();

  // If the player's score is greater than the lowest high score for this
  // skill level, ask them for their name.
  if ( ComputeScore( pGameState ) >
       (*hScores)->Score[pGameState->sSkillLevel - 1][9].lScore )
  {
```

```
    // Create the application modal dialog.  HandleHighScoresEvent is the
    // function we want to receive events for this dialog, and
    // SetModalDialogMenuState() will disable and enable the menus as
    // appropriate.  We also specify the enter/return keys should be
    // the same as clicking on the PB_NAME_OK button item, and the escape/
    // tilde keys be the same as clicking on the PN_NOT_INTERESTED button.
    pDialog = ConstructAppModalDialog( DLG_HIGH_SCORE,  HandleHighScoresEvent,
                                       nil, SetModalDialogMenuState,
                                       PB_NAME_OK, PB_NOT_INTERESTED, 0, 0 );

    // Initialize the dialog by setting the text edit field item EF_NAME to
    // the name last typed in by the player.  Also, select it.
    SetIText( (Handle) GetItemHandle( pDialog, EF_NAME ),
              (Pstr) (*hScoreName)->szName );
    SelIText( pDialog, EF_NAME, 0, 99 );

    // Show the dialog.
    ShowWindow( pDialog );
  }
}
```

The following function handles the events for the dialog box, and can also be found in **Scores Window.c**.

```
static void HandleHighScoresEvent( short       sItemHit,
                                   DialogPtr   pDialog,
                                   EventRecord *pEvent )
{
  Str255  str255;
  short   sLength;

  switch ( sItemHit )
  {
    // If the user hit the OK button...
    case PB_NAME_OK:

      // Get the name they entered.
      GetIText( (Handle) GetItemHandle( pDialog, EF_NAME ), str255 );
      sLength = (short) *str255;

      // If no name was entered, or it was too long, display an alert to
      // tell them so.  Select the name text to make it easier for the
      // player to correct.
```

```
        if ( ( sLength < 1 ) ||
             ( sLength > NAME_LENGTH ) )
        {
          SelIText( pDialog, EF_NAME, 0, 99 );
          ShowAlert( DLG_BAD_NAME, nil, nil, SetModalDialogMenuState );
        }

        // If the name was okay...
        else
        {
          // Close the application modal dialog.
          DestructAppModalDialog( SetModalDialogMenuState );

          // Save the name so we can default to it next time.
          memcpy( (*hScoreName)->szName, str255, sLength + 1 );

          // Add the name to the high scores list.
          AddHighScore( str255, sLength );
          // If the high scores window is up, update it to reflect the new
          // high score.
          if ( pDialogScores )
            ViewNewLevel( sSkillLevelViewed );
        }
        break;

      // If the user hit the "Not Interested" (cancel) button...
      case PB_NOT_INTERESTED:

        // Close the application modal dialog.
        DestructAppModalDialog( SetModalDialogMenuState );
        break;
    }
}
```

Using the preceding explanations and examples, and including the **App Modal Dialog.c** unit in your game should make it fairly easy for your game to support application modal dialog boxes. It's well worth the effort.

Supporting Application Modal Alerts

Alerts provide a very useful function. They display a message to the user and solicit a simple yes/no type response. But, as in the case of modal dia-

log boxes, they prevent the user from interacting with other applications running on the Macintosh. It would be a shame to lose the simplicity of these alerts. However, if your game supports application modal dialog boxes as was just described, you need very little additional coding in order to support application modal alerts (these that prevent the user from interacting with any other windows in your game, but allow the user to interact with windows belonging to other applications).

Instead of using the toolbox `Alert()` routine, you can call the `ShowAlert()` routine defined in **Common.c**, which takes four parameters. The first parameter is the resource ID of the alert to load. Instead of defining a resource of type 'ALRT', though, define the alert like a regular dialog box of type 'DLOG'. So, in essence, the first parameter is a dialog box resource ID. The second parameter is a rectangle in which to center the alert when it's drawn. You can pass `nil` for this parameter to have the alert centered on the screen. The third parameter is a pointer to a function that you want called when the user dismisses the alert. This callback function takes one parameter, a short integer denoting the item ID that caused the alert to be dismissed. A value of `nil` is allowed and means that no function will get called when the alert is dismissed. Use `nil` when no action is needed after the alert goes away, such as when the alert is simply displaying information and an OK button. The final parameter is a pointer to a function that handles the enabling and disabling of menu items when an application modal dialog box (or, in this case, an application modal alert) is active. This callback function was described above in the section on supporting application modal dialog boxes. You can specify `nil` for this parameter if you don't need to disable any menu items when an application modal alert is shown (or if you have already done so before calling this routine).

```
// This private variable is used to store the address of the callback
// function supplied to ShowAlert(). This callback function, if specified,
// will be called when the alert is dismissed by the user. The callback
// function will be provided with the ID of the item which caused to alert to
// be dismissed, which is exactly what you'd get if you call the Alert()
// toolbox routine.
static void     (*pfnHandleResult)( short );

void ShowAlert( short  sAlertID,
```

```
                Rect    *rectCenterOn,
                void    (*pfnHandleResultCallback)( short ),
                void    (*pfnHandleMenus)( Boolean ) )
{
  DialogPtr  pDialog;

  // If no rectangle was specified for centering the alert on, default to the
  // center of the screen.
  Rect       rectToCenterOn = screenBits.bounds;

  // If a rectangle was specified for centering the alert on, use it.
  if ( rectCenterOn )
    rectToCenterOn = *rectCenterOn;

  // Since only one alert can be on the screen at one time, we don't need an
  // array to save the result callback function.  We can use a single private
  // variable instead.
  pfnHandleResult = pfnHandleResultCallback;

  // Load an application modal dialog using the set of routines described
  // above.  All alerts use the common HandleAlertEvent() routine to handle
  // events for that alert.
  pDialog = ConstructAppModalDialog( sAlertID,  HandleAlertEvent, nil,
                                     pfnHandleMenus, PB_OK, 0, 0, 0 );

  // Center the alert, either on the screen, or within the rectangle
  // specified as a parameter to this function.
  CenterWindow( &pDialog->portRect, rectCenterOn, true, pDialog );

  // Beep to signify that this is an alert.
  SysBeep( 1 );

  // Show the alert and make it the frontmost window.
  ShowWindow( pDialog );
  SelectWindow( pDialog );
}
```

The `HandleAlertEvent()` routine handles events for alerts. Whenever a dialog box event for this alert occurs, this routine gets called. Its main purpose is to determine if the user has dismissed the alert and if so, to call the routine you specified when the alert was created to handle the return code of the alert (which is the item ID used by the user to dismiss the alert). This code can be found in **Common.c**.

```
static void HandleAlertEvent( short       sItemHit,
                              DialogPtr   pDialog,
                              EventRecord *pEvent )
{
  short   sItemType;
  Handle  hItem;
  Rect    rect;

  // Get the item information on the item clicked on by the user.
  GetDItem( pDialog, sItemHit, &sItemType, &hItem, &rect );

  // If the item was a push button (or a resource control), the user is
  // dismissing the alert.  We need to check the item type since the user
  // could click on the alert's icon or message, which would be passed to
  // this routine since they are valid clicks on a dialog item.  Clicks on
  // those item types can be ignored.
  if ( ( sItemType == ctrlItem ) ||
       ( sItemType == ctrlItem + resCtrl ) )
  {
    // Close the application modal dialog representing this alert.
    DestructAppModalDialog( SetModalDialogMenuState );

    // If a callback function was specified, call it and pass it the ID of
    // the item the user used to dismiss this alert.
    if ( pfnHandleResult )
      (*pfnHandleResult)( sItemHit );
  }
}
```

You need only the two previous routines to support application modal alerts (assuming, of course, that you are supporting application modal dialog boxes). Here's an example of code your would write to display an application modal alert using the `ShowAlert()` routine from **Offscreen Graphics.c**. Most of the routine is left out so that we can focus on the `ShowAlert()` call.

```
void CheckMonitorColors( Boolean bReloadGraphics )
{
  // Show an application modal alert to the user.  The DLOG resource id is
  // NOT_ENOUGH_MEMORY_ALERT_ID.  nil is passed as the center on parameter so
  // that the alert gets centered on the screen.  When the user dismisses the
  // alert, we want the HandleMemoryAlertResult() function to get called.  it
  // will be passed the item id which was used to dismiss the alert.  Finally,
```

```
  // the SetModalDialogMenuState() will be called to disable and enable menu
  // items to reflect the fact that an application modal alert is on the
  // screen.
  ShowAlert( NOT_ENOUGH_MEMORY_ALERT_ID, nil,
             HandleMemoryAlertResult, SetModalDialogMenuState );
}
```

Here's the a routine that handles the alert's result code. It can be found in **Offscreen Graphics.c**.

```
static void HandleMemoryAlertResult( short sResult )
{
  // Show the Desert Trek main window.
  ShowTrekWindow();

  // If the user clicked on the "How Do I Fix This?" button, show another
  // application modal alert.  We specify no routine to handle the result
  // code since this alert is for information only (it has only one button to
  // dismiss it).
  if ( sResult == PB_MORE_INFO )
    ShowAlert( MEMORY_HELP_ALERT_ID, nil, nil, SetModalDialogMenuState );

  // If the user doesn't want any more warnings, set the flag to disable them.
  else if ( sResult == PB_DISABLE_WARNINGS )
    **hbColorDepthWarning = false;
}
```

CHAPTER 7

QUICKDRAW

QuickDraw is probably the most fundamental Macintosh toolbox manager. Anything that the user sees on the screen is a direct result of one of the QuickDraw toolbox routines. This includes text as well as graphics. Other toolbox managers use QuickDraw when they need to draw something on the screen. For example, the window manager uses QuickDraw to draw all window parts such as title bars and title text. The menu manager uses QuickDraw to draw the menu bar and to highlight and unhighlight menu items as the user selects them.

Because QuickDraw is so fundamental to the operation of the Macintosh, it is easily one of the largest and most complex toolbox managers. Anyone new to Macintosh programming can easily become overwhelmed with the task of trying to learn everything there is to know about QuickDraw. In fact, there are entire books whose sole purpose in life is to describe QuickDraw, or even just a specific subset of QuickDraw. Fortunately, the core functions of QuickDraw, and those functions necessary to write a complete game, can be briefly covered in much less space. This chapter will take you through the fundamentals of QuickDraw, explaining how drawing occurs on the Macintosh screen, within specific windows, or offscreen. You will learn how to draw graphic elements and text as well as see a few tricks used to provide special graphics effects in your game.

Points, Rectangles, and Regions

When drawing, you need to specify not only what to draw, but where to draw it. The coordinate system that QuickDraw uses specifies points on the screen using a horizontal and vertical coordinate. Because you will frequently need to use point coordinates in your game, the toolbox defines a point structure for you. We have already seen the `Point` type as it relates to mouse down events.

```
struct Point {
              short v;  // Vertical coordinate.
              short h;  // Horizontal coordinate.
};

typedef struct Point Point;
```

The horizontal coordinate, `h`, specifies how many pixels the point lies from the left edge of the screen. The vertical coordinate, `v`, specifies how many pixels the point lies from the top of the screen. This means that the origin of the Macintosh coordinate system is the upper left of the screen.

QuickDraw also defines a data structure named `Rect`. You will find this a most useful type and use it frequently throughout your program. The `Rect` data structure defines a bounding rectangle, which has top, left, bottom, and right edges. See Figure 7.1 for a pictorial view of the Macintosh coordinate system.

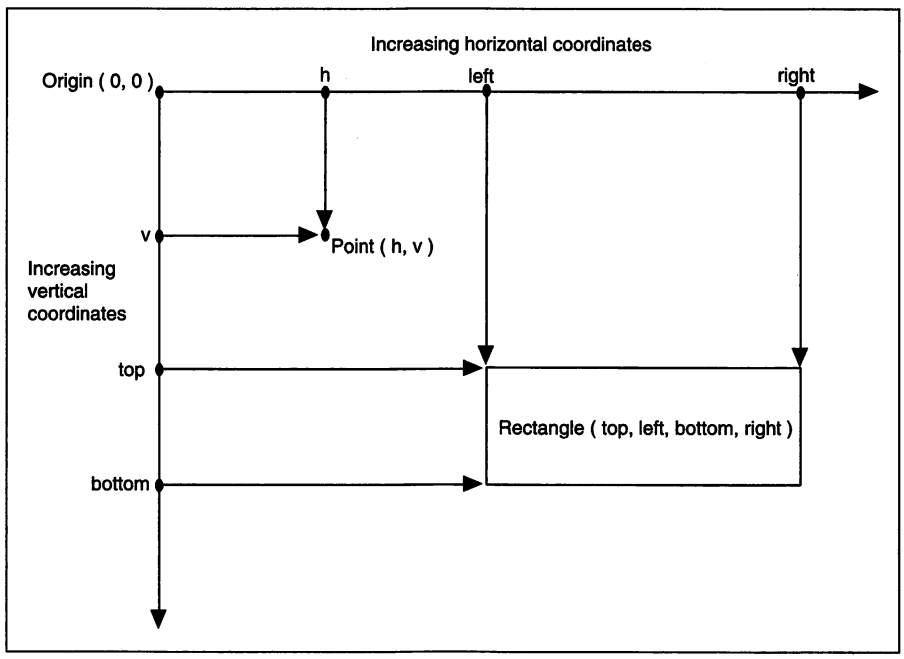


Figure 7.1 The Mac coordinate system.

```
struct Rect {
              short top;        // Top edge of the rectangle.
              short left;       // Left edge of the rectangle.
              short bottom;     // Bottom edge of the rectangle.
              short right;      // Right edge of the rectangle.
};

typedef struct Rect Rect;
```

Many QuickDraw toolbox calls use bounding rectangles. This makes sense because if you look at the windows on a Macintosh screen, the contents regions of those windows are rectangles. Look a little more closely, and you'll see that just about every part of a window or control constitutes a rectangle. The title bar, close box, zoom box, and size box of a window are all rectangles. The up arrow, down arrow, page up region, page down region, and thumb of a scroll bar control are all rectangles too. Rectangles are not only used to define actual rectangles, but other objects as well. When you want to define a circle or oval, you describe that object to QuickDraw by specifying the rectangle that bounds the circle (which would be a square) or oval. Even standard button controls are considered rectangles with rounded corners (and thus are called rounded rectangles). Frequently, seemingly complex objects can be simply described using a number of smaller rectangles. As we look at the QuickDraw toolbox routines, we'll see the rectangle at work in many roles.

Finally, what happens if an object you're describing really can't be specified by using a single rectangle. In this case, you'd use a structure of type `Region`. Regions are much more complex and difficult to use than rectangles, but they do provide a way to describe objects that can't be described with rectangles alone. Regions need to be built and are frequently referenced by using a handle to that region (because region definitions can be of varying size). However, if you look at the region data types given as follows, you'll notice that one field of the region data structure is a rectangle, which is the smallest rectangle that encloses the entire region. Rectangles really are pervasive in QuickDraw. I'm not going to spend much more time on regions, because they typically do not play an important role in games. In fact, Desert Trek does not use any regions at all.

```
struct Region {
                short rgnSize;  // Size of the region in bytes
                Rect rgnBBox;   // Region's bounding rectangle
};

typedef struct Region Region;
typedef Region *RgnPtr, **RgnHandle;
```

Operations on Points

The Macintosh toolbox provides many functions that operate on points. They range from setting points to combining points to translating points from one coordinate space to another. We have already seen the `GlobalToLocal()` and `LocalToGlobal()` toolbox functions described in Chapter 4 on Working with windows. Both of these functions convert points from one coordinate space to another. The following functions also operate on points. Note that not all toolbox calls that operate on points are listed, though the ones you would most likely use in your own game are shown.

```
// Sets a Point structure to the specified coordinates.
void SetPt(       Point      *pPoint,
                  short      sHCoordinate,
                  short      sVCoordinate );

// Returns true if the two points are equal.
Boolean EqualPt(  Point      pt1,
                  Point      pt2 );

// Adds two points together.
void AddPt(       Point      ptToAdd,
                  Point      *pptToAddTo );

// Subtracts one point from another.
void SubPt(       Point      ptToSubtract,
                  Point      *pptToSubtractFrom );

// Returns true if the given point lies within the given
// rectangle.  This call is very useful for determining
// if mouse clicks occur within objects contained in your
// windows and dialogs.
Boolean PtInRect( Point      pt,
                  Rect       *pRect );
```

Operations on Rectangles

The Macintosh toolbox also provides an almost dizzying array of functions to manipulate rectangles. The following are some of the more useful ones. Because a rectangle structure is 8 bytes in length, all toolbox calls take pointers to rectangles as parameters even if the rectangle isn't going to be modified by the call.

```
// Sets a Rect structure to the specified boundary.
void SetRect(       Rect        *pRect,
                    short       sLeft,
                    short       sTop,
                    short       sRight,
                    short       sBottom );

// Sets a Rect structure given two points (the diagonal of
// the rectangle).
void Pt2Rect(       Point       point1,
                    Point       point2,
                    Rect        *pRect );

// Returns true if the two rectangles are equal.
Boolean EqualRect(  Rect        *pRect1,
                    Rect        *pRect2 );

// Moves a rectangle by the amount specified.  The horizontal
// offset is added to the rectangle's left and right edges,
// and the vertical offset is added to the rectangle's top
// and bottom edges.  Positive offsets move the rectangle
// down or to the right.  Negative offsets move the rectangle
// up or to the left.
void OffsetRect(    Rect        *pRect,
                    short       sHOffset,
                    short       sVOffset );

// Changes the dimensions of a rectangle.  The horizontal
// inset is added to the left edge, and subtracted from the
// right edge.  The vertical inset is added to the top edge,
// and subtracted from the bottom edge.  A positive inset
// shrinks the rectangle.  A negative inset expands the
// rectangle.
void InsetRect(     Rect        *pRect,
                    short       sHInset,
```

```
                    short       sVInset );

// Computes the smallest rectangle that completely encloses
// the two given rectangles.
void UnionRect(     Rect        *pRect1,
                    Rect        *pRect2,
                    Rect        *pRectUnion );

// Returns true if the two rectangle intersect.  The
// rectangle formed by the intersection is returned in
// pRectIntersection.
Boolean SectRect(   Rect        *pRect1,
                    Rect        *pRect2,
                    Rect        *pRectIntersection );
```

Operations on Regions

Again, regions are objects that games often do not need to use. However, if you want to use a toolbox routine that requires a region handle, you'll need to create a region, define that region, and destroy that region once you're finished with it. I won't go into how to define a complex region here, but using the following toolbox calls, you can define a region whose boundary is a simple rectangle.

```
// Creates a new region and returns a handle to it.
RgnHandle NewRgn( void );

// Defines a region's boundary to be that of the specified
// rectangle.
void RectRgn(  RgnHandle     hRgn,
               Rect          *pRect );

// Disposes of a region, freeing up the memory taken by that
// region.
void DisposeRgn( RgnHandle hRgn );
```

Graphics Ports

When you draw something using QuickDraw, where does that drawing actually take place and get displayed? Is it always directly to the screen?

Well, that doesn't really make much sense. Who knows where the user might have moved your game window, or how many other windows the user has opened on the Macintosh screen. If your game had to take all this into account whenever you wanted to draw something, it'd be a miracle if everything went where it was supposed to. QuickDraw provides a simple solution to this problem: All drawing takes place in the current graphics port. A graphics port is a QuickDraw entity that defines a location for drawing, be it on the screen or offscreen. A graphics port defines not only a drawing space, but all kinds of characteristics for that drawing space. These characteristics define how drawing takes place within that graphics port.

A graphics port's characteristics are stored in a record just like window information is stored in a window record and control information is stored in a control record. In fact, we saw in Chapter 4 on Working with windows that the first element of a window record is a graphics port record. This means that window pointers are in essence graphics port pointers, and can be used wherever you can use graphics ports. When your game creates a window, it also creates a graphics port associated with that window, causing all drawing to that graphics port to take place within that window.

Again, the individual fields of a graphics port record will not be listed because you should never access these fields directly. There are a number of toolbox calls that allow you to read and set the characteristics of a graphics port's record. However, there is a distinction between color-capable graphics ports and "normal" graphics ports. For several years, the Macintosh supported only black-and-white displays. When the original QuickDraw routines were designed, little color support was defined into graphics ports. After the advent of color-capable Macintoshes, the definition of a graphics port needed to be changed to support the new color capabilities. However, the new definition needed to be backwards compatible so that older programs written using the original graphics ports could continue to be used. To accomplish this, a new color-capable graphics port record was defined, leaving the original graphics port unaltered. However, all QuickDraw toolbox routines were implemented such that they can take pointers to either type of graphics port, color or original. This means that your game will not need to be

concerned by the differences in the two graphics port record types. Creating, manipulating, and destroying graphics ports are identical for either type. The following toolbox definitions show how graphics port records and pointers are defined.

```
struct GrafPort {
     .
     BitMap portBits,              // The port's bitmap, which
     .                             // we'll see in a moment.
     .
};

struct CGrafPort {
     .
     PixMapHandle portPixMap,      // The handle to a color
     .                             // bitmap, which we'll see a
     .                             // little later.
};

typedef struct GrafPort GrafPort;
typedef GrafPort *GrafPtr;
typedef struct CGrafPort CGrafPort;
typedef CGrafPort *CGrafPtr;
```

Offscreen Graphics Ports

Because graphics ports for windows are automatically created with the window, when would you need to create any graphics ports yourself? All drawing occurs in windows, or at least somewhere on the screen, doesn't it? Well, if the topic of this section isn't hint enough, the answer is no, not all drawing occurs directly on the screen. This is especially true for games. Frequently you will want to perform all drawing operations off-screen, and then transfer the complete offscreen image directly to your game window in one shot. This has several advantages. First, the user won't see the individual drawing commands used to create the complete image actually take place. Second, constant drawing directly to a window graphics port can cause flicker because you often need not only to draw an object at its current position, but you need to erase that object from its

previous position. Third, if your game has objects that move over a background, that background needs to be stored somewhere in its original state (without any objects over it). Fourth, the complex objects that you draw need to be stored somewhere. You don't want to be loading them from the resource fork every time you need them. A great way to store them is to use an offscreen graphics port, from where they can be copied anytime you need them. There are plenty of other reasons to use offscreen graphics ports, but let's sum it up by saying that just about anything that appears on the screen for game programs is first drawn in an offscreen graphics port.

Bitmaps

Graphics port records contain characteristics about the port they describe, and one of these fields is the port's bitmap. It is the port's bitmap that actually contains the image. When you perform drawing commands to a graphics port, the results of that drawing is reflected in the bitmap. The graphics port's bitmap for a window is the content region of that window. When you draw to the window's graphics port, its bitmap is affected, and those changes are automatically displayed on the screen in the content region of that window. An offscreen port's bitmap, however, resides only in memory, not on the screen. For this reason, offscreen bitmaps are often referred to as *in-memory* bitmaps. The important thing to realize here is that you need to allocate the memory needed by an offscreen port's bitmap. The amount of memory needed for an offscreen bitmap depends on two factors: the size of the bitmap and the number of colors the bitmap supports (often referred to as the number of bit planes). The size of a bitmap is often expressed in pixels, where a pixel represents a single dot. Many standard Macintosh monitors support 640x480 pixels, meaning that the monitor displays 640 pixels left to right and 480 pixels top to bottom. The number of colors a bitmap supports is 2 raised to the power of the number of planes in a bitmap. For example, a black-and-white bitmap, which supports just one color (either white on a black background or black on a white background), has a plane count of 1 ($2^1 = 1$). A bitmap that supports 256 colors, a common number chosen by many games, requires 8 bit planes ($2^8 = 256$).

To compute the amount of memory you need to allocate for an offscreen bitmap, you need to know the width and height of the bitmap, as well as the number of colors. However, there's a little catch here. The in-memory representation of a bitmap has a certain restriction: Each row of a bitmap must be word aligned. What does that mean? First, a "row" of the bitmap can be considered a single line of that bitmap. In other words, a row has a height of 1 pixel and runs the entire width of the bitmap. So, if a bitmap has a height of 12 pixels, you can consider that bitmap as having 12 rows. When I say word aligned, I mean that a row of the bitmap must start on a word address, which is a 16-bit value. This means that the smallest amount of memory required by a row in a bitmap is 16 bits, since the next row of the bitmap must start on the next word boundary. The number of bits required for a row of a bitmap is the width of the bitmap times the number of color planes. Thus, a bitmap 12 pixels wide and containing only 1 bit plane (meaning that this is a black-and-white bitmap) requires 16 bits, or 2 bytes, for every row. Notice that in this example, 4 bits of every row go unused. Once you compute the number of bytes required by a row in a bitmap, simply multiply that number by the number of rows in the bitmap to determine how much memory that bitmap requires. Remember, the number of rows in a bitmap is the height of the bitmap expressed in pixels.

Example

So, how many bytes would a black-and-white bitmap 46 pixels wide by 15 pixels high take? A single row in this bitmap needs 46 bits (46 pixels times 1 bit plane)..To make each row word aligned, however, a single row needs to be padded to the next 16-bit boundary after 46, which is 48. That's 3words (48 divided by 16), or 6 bytes (48 divided by 8). Thus, the number of bytes required by this bitmap is 6 bytes per row times 15 rows, or 90 bytes. If your game needed to use an offscreen graphic ports with the properties described in this example, you would need to allocate 90 bytes for the bitmap contained in that graphics port.

Just like many other entities on the Macintosh, a bitmap is described by a bitmap record. The bitmap record contains only three fields: a pointer to the memory used by the bitmap, the number of bytes in each row of the

bitmap, and the bounding rectangle of that bitmap (which is used to specify the dimensions of a bitmap). You need to set all three values yourself when creating an offscreen bitmap. Note that the memory taken by a bitmap is nonrelocatable, meaning that it is fixed in memory. You might want to keep that in mind when allocating memory for bitmaps.

```
struct BitMap {
    Ptr     baseAddr;     // Pointer to the bitmap's memory.
    short   rowBytes;     // Number of bytes in each row.
    Rect    bounds;       // Bounding rectangle.
};
```

A color-capable bitmap requires more information than that required by a monochrome bitmap. In order to distinguish a color-capable bitmap from a monochrome bitmap, color-capable bitmaps are referred to as *pixmaps*, or *pixel maps*. Thus, a pixmap is the color equivalent of a bitmap. However, in order to make pixmaps as similar as possible to bitmaps, the first three fields of a pixmap record are exactly the same as the three fields of a bitmap record. So, when creating a pixmap, you still allocate the storage taken by that pixmap in the same way you do for bitmaps. However, you also now need to allocate the pixmap record (yes, another memory allocation!). The Macintosh provides the following two routines to allocate and de-allocate a pixmap record. Unless otherwise noted, when I refer to bitmaps in the future, you can assume that I'm also referring to pixmaps.

```
// Creates a new pixmap record and returns the handle to that
// pixmap to you.  Make sure to dereference the handle when
// assigning values to the baseAddr, rowBytes, and bounds
// fields (you need to assign these values yourself, just as
// you would for a bitmap).
PixMapHandle NewPixMap( void );

// Disposes of a pixel map record.  Make sure to free up the
// memory taken by the image of the pixmap, which you
// allocated yourself.
void DisposePixMap( PixMapHandle hPixMap );
```

You need to create and destroy bitmaps for offscreen graphics ports yourself. When a graphics port is first initialized, the bitmap associated with that graphics port is the Macintosh screen itself. This means that if you started drawing to that graphics port, the drawing would go directly to the screen. This, of course, is totally undesirable because you should never draw directly to the screen's bitmap. That would affect other application's windows as well as your own. It's not the Macintosh way. So, in essence, this means that creating an offscreen graphics port does not create a bitmap that you can use to hold that port's image. You'll need to create an offscreen bitmap and assign that bitmap to the graphics port in question when created, and free up the memory taken by that bitmap when the port is destroyed. We'll see how to do that in a minute.

Creating and Destroying Graphics Ports

You must allocate the storage for a graphics port record yourself because the Macintosh does not do it automatically for you. A couple of the fields contained in the graphics port record contain pointers to additional data. The Macintosh will allocate and maintain the memory used by these fields for you. The memory allocated for a graphics port's record must be nonrelocatable, or fixed. Keep this in mind when allocating storage for the port's record. After allocating memory for a graphics port, use the following toolbox call to initialize that port and have the Macintosh allocate the internal structures needed for that port.

```
// Initializes a new graphics port.  The port is made the
// current graphics port.
void OpenPort( GrafPtr pGraf );

// Initializes a new color graphics port.  The port is made
// the current graphics port.
void OpenCPort( CGrafPtr pCGraf );
```

If you want to restore a graphics port to its initialized state, use the following toolbox call. This call is appropriate to use if you've changed the

drawing properties of a graphics port and want to quickly get back to the defaults using a single toolbox call.

This call does not allocate memory for the port's internal structures, and thus cannot be used on a port that hasn't been first initialized using the `OpenPort()` routine.

```
// Reinitializes an existing graphics port.  The port is made
// the current graphics port.
void InitPort( GrafPtr pGraf );

// Reinitializes an existing color graphics port.  The port
// is made the current graphics port.
void InitCPort( CGrafPtr pCGraf );
```

After you have finished with a graphics port, you need to destroy it. The following toolbox call closes a port and releases all memory taken by the port's internal data structures. Keep in mind that you need to free up the memory taken by the graphics port's record and associated bitmap.

```
// Closes the specified graphics port.  Memory used by
// internal structures is freed, but not the memory used
// by the port's record itself.
void ClosePort( GrafPtr pGraf );

// Closes the specified color graphics port.  Memory used by
// internal structures is freed, but not the memory used
// by the port's record itself.
void CloseCPort( CGrafPtr pCGraf );
```

We will see an example of how to create and destroy a graphics port in a moment.

Associating Bitmaps with Graphics Ports

Remember, creating and initializing a graphics port does not provide that port with a bitmap in which to draw. One of the first things you need to do after creating a graphics port is to associate a bitmap you've created

with that port. After creating your bitmap and opening the graphics port, use the following toolbox call to associate the two. Notice that like many of the QuickDraw toolbox calls, you do not specify the graphics port you want to affect. This means that the toolbox call operates on the current graphics port, and you should make sure the graphics port you want affected is the current graphics port.

```
// Associates a bitmap with the current graphics port.
void SetPortBits( BitMap bitMap );

// Associates a pixmap with the current color graphics port.
void SetPortPix( PixMapHandle hPixMap );
```

Setting and Getting the Current Graphics Port

As already mentioned, most QuickDraw toolbox operations affect the current graphics port. You need a way to change what graphics port is current in order to draw to a different port. You'll also need to obtain the current graphics port, mainly so that you can set the current graphics port back to what it was before you changed it. The following two toolbox calls allow you to obtain and set the current graphics port. Note that the following two routines work just as well for color graphics ports, in which case you'd simple supply a `CGrafPtr` instead of a `GrafPtr`.

```
// Obtains the current graphics port.
void GetPort( GrafPtr *ppGraf );

// Set the current graphics port to the one specified.
void SetPort( GrafPtr pGraf );
```

Setting a Port's Clipping Region

When updating windows on the screen, the Macintosh automatically clips any drawing needed to just the part of the window that needs to be

drawn. This was accomplished by using the `BeginUpdate()` and `EndUpdate()` toolbox calls. Recall that a window's clipping region is the only part of the window that actually gets drawn to. Anything you try to draw outside the clipping region gets clipped, meaning that the area outside the clipping region remains unaffected. You can get and set a graphics port's clipping region using the following toolbox calls. Notice that `ClipRect()` doesn't require a region handle, and thus provides an easy way to set a port's clipping region. Also note that these calls do not take a graphics port pointer as a parameter, meaning that they affect the current graphics port. Make sure that you've set the current graphics port to the one you wish to get or set the clipping region for.

```
// Gets the current ports clipping region.  Notice that the
// handle itself does not change (you don't pass in a pointer
// to the handle), but its master pointer will get set to
// point to the port's clipping region.
void GetClip( RgnHandle hRgn );

// Sets the current port's clipping region.  The Macintosh
// copies the region you supply and uses that copy.
void SetClip( RgnHandle hRgn );

// Sets the current port's clipping region to the rectangle
// specified.
void ClipRect( Rect *pRect );
```

QuickDraw Globals

When you initialize the QuickDraw toolbox manager, several global variables are made available to your program. These global variables can be used throughout your code for various purposes. A couple of the global variables you might find useful are as follows:

```
// This global variable is used when calling InitGraf to
// initialize the QuickDraw toolbox manager.  You can
// reference this variable anytime to see what the current
// graphics port is, but you should really use GetPort()
// instead.
GrafPtr qd.thePort;
```

```
// The bitmap referencing the Macintosh screen.  This bitmap
// is useful for determining the size of the screen your game
// is running on.  Desert Trek references this global
// variable frequently when centering windows and dialogs on
// the screen.
BitMap qd.screenBits;
```

Offscreen Graphics Port Example

The following code fragment from **Scores Window.c** creates an offscreen graphics port for the high scores window when the monitor is running in black-and-white mode. Color graphics support in Desert Trek does not use the somewhat cumbersome process of allocating and maintaining color graphics ports and pixmap records. In a moment, we'll see a much easier way to support offscreen color graphics ports.

```
static GrafPtr pGrafScores = nil;
static BitMap  bitmapScores;
static Rect    rectScores;

void ConstructScoresWindowOffscreen( void )
{
  GrafPtr   pGrafCurrent;
  short     sBitmapRowBytes;
  Size      sizeBitmap;

  // Save the current graphics port so that we can restore it after creating
  // the offscreen graphics port.
  GetPort( &pGrafCurrent );

  // Allocate storage for the graphics port record.
  pGrafScores = (GrafPtr) NewPtr( sizeof( GrafPort ) );

  // Initialize (open) the port for the first time.  OpenPort() also sets the
  // given port to be the current graphics port.
  OpenPort( pGrafScores );

  // Compute the number of bytes needed for a single row of the bitmap we plan
  // to associate with the offscreen graphics port.  Remember, the rows need
  // to be word aligned, so round up to the nearest word.
```

```
  sBitmapRowBytes = (((rectScores.right - rectScores.left - 1) / 16) + 1) * 2;

  // Compute the number of bytes needed for the entire bitmap, which is the
  // number of bytes needed per row times the number of rows.
  sizeBitmap = ( rectScores.bottom - rectScores.top ) * sBitmapRowBytes;

  // Allocate the storage needed for the in-memory bitmap, and set the bitmap
  // fields accordingly.
  bitmapScores.baseAddr = (QDPtr) NewPtr( sizeBitmap );
  bitmapScores.rowBytes = sBitmapRowBytes;
  bitmapScores.bounds = rectScores;

  // Associate the new bitmap with the new graphics port.
  SetPortBits( &bitmapScores );

  // Set the current graphics port back to what it was before this function
  // was called.
  SetPort( pGrafCurrent );
}
```

Once the graphics port and its bitmap are no longer needed, the following code is called to release the memory taken up by the bitmap and the graphics port record. This function can be found in its entirety in **Scores Window.h**.

```
void DestructScoresWindowOffscreen( void )
{
  // If the graphics port is a valid pointer...
  if ( pGrafScores )
  {
    // Free the memory allocated for the bitmap.
    DisposPtr( (Ptr) bitmapScores.baseAddr );

    // Close the graphics port.  This free up memory the Macintosh allocated
    // for the port's internal structures.
    ClosePort( pGrafScores );

    // Free the memory allocated for the graphics port's record.
    DisposPtr( (Ptr) pGrafScores );

    // Set the graphics port pointer to nil, so we know that it's invalid.
    pGrafScores = nil;
  }
}
```

Offscreen Graphics Worlds

Maintaining an offscreen graphics port is somewhat complicated. Not only must you allocate a graphics port record, but you also need to set up the offscreen bitmap you want to associate with that port. Computing the offscreen bitmap's memory requirement isn't trivial. After finishing with an offscreen graphics port, you need to clean up the memory used by the bitmap and graphics port record. Adding color support adds an additional element to allocate and maintain: the pixmap record and handle. Last, the memory allocated for a graphics port record and bitmap need to be fixed, causing potential memory fragmentation problems.

You're probably saying, "there's got to be an easier way." Fortunately, yes, there is an easier way to maintain offscreen graphics environments. The Macintosh provides graphics worlds, which make the maintenance of offscreen graphics environments much simpler than the process just described. Then why did I bother to tell you about graphics ports and bitmaps in the first place? Well, for two reasons. First, offscreen graphics worlds still use graphics ports and bitmaps. It's just that they allocate and maintain the memory used by graphics ports and bitmaps for you. You still need to know about their existence and how to use them. Second, older Macintoshes that don't support 32-bit color QuickDraw do not support graphics worlds. This means that if your game is going to run on such a Macintosh, it needs to create and maintain graphics ports and bitmaps itself. Desert Trek makes a "compromise" in this area. Black-and-white support for Desert Trek is accomplished using the manual method of creating and maintaining graphics ports and bitmaps, while color support is accomplished via the use of graphics worlds. This means that in order to run Desert Trek in color, your Macintosh must support 32-bit color QuickDraw (color-capable Macintoshes without 32-bit color QuickDraw are rare, but they could still run Desert Trek in black-and-white mode).

To be honest, graphics worlds are fairly complex and allow you to define a large number of characteristics for the drawing environment. However, you can easily use just the bare-bones features of graphics worlds and still get everything you need for most game purposes. When implemented correctly, you rarely need to distinguish between the use of

graphics worlds or graphics ports and bitmaps. The biggest difference comes when creating and destroying graphics worlds, which is a lot simpler than creating and destroying graphics ports and bitmaps.

To use the graphics world toolbox calls from most compilers you need to explicitly include Apple's include file **QDOffscreen.h** in your code, which contains all the definitions and function prototypes needed to use offscreen graphics worlds. These definitions are not automatically included in your game project for you.

Graphics world pointers are used when calling toolbox routines that deal with graphics worlds. Their definition is exactly that of a color graphics port. However, note that you do not need to allocate storage for the graphics port record when using graphics worlds. The Macintosh automatically takes care of that for you.

```
typedef CGrafPtr GWorldPtr;
```

Creating an Offscreen Graphics World

To create an offscreen graphics world, use the following toolbox call.

```
// Creates an offscreen graphics world.  Specify a pointer
// to the graphics world pointer, which will get set for you.
// Also specify the pixel depth of the offscreen graphics
// world (which can be 0, 1, 2, 4, 8, 16, or 32), and the
// bounding rectangle.  You can specify nil for hCTab and
// hDC, and 0 for fGWorld.  These parameters are optional.
QDErr NewGWorld( GWorldPtr        *ppGWorld,
                 short            sPixelDepth,
                 Rect             &pRectBounds,
                 CTabHandle       hCTab,
                 GDHandle         hGD,
                 GWorldFlags      fGWorld );
```

This toolbox routine returns to you a graphics world pointer in the parameter `ppGWorld`. You need to supply the bounding rectangle of the offscreen graphics world in `pRectBounds` and the number of bit planes in

sPixelDepth. The remaining parameters are optional and won't be described in detail here. You should simply supply nil for hCTab (the handle to a color table) and hGD (the handle to a graphics device), and 0 for fGWorld (the graphics world creation flags). This routine will allocate all the necessary memory needed for the graphics port and offscreen bitmap. So, even though the toolbox call looks a little scary, it really saves you a lot of coding.

Setting the Current Graphics World

Remember, you need to use SetPort() to set the current graphics port to the particular port in which you want to draw. There's a similar toolbox routine that you need to use concerning the current graphics world. Make sure to use the following toolbox call to set the current graphics world to the desired graphics world.

```
// Sets the current graphics world.  You can specify nil for
// the hGD parameter when setting the current graphics world
// to one that you've created.  If you're setting the
// graphics world back to one that you' previously queried
// with GetGWorld(), pass the hGD you got from that call.
void SetGWorld(  GWorldPtr      pGWorld,
                 GDHandle       hGD );
```

You'll need to query the current graphics world before changing it so that you can set it back when you're finished drawing to that other graphics world. This is very similar to using GetPort() and SetPort() for getting and setting the current graphics port.

```
// Obtains the current graphics world and device handle.
void GetGWorld(  GWorldPtr      *ppGWorld,
                 GDHandle       *phGD );
```

This routine returns a device handle as well as a graphics world pointer. All you need to know about the device handle is that you should pass it back to SetGWorld() when you restore the current graphics world back to what its original state.

Locking a Graphics World's Pixmap

To draw to an offscreen graphics world, you simply need to make it the current graphics port by using `SetGWorld()`. However, there's one catch: Remember that the bitmap (or pixmap) you manually allocate for a graphics port is fixed in memory. When drawing to such a bitmap, you don't need to worry about the Macintosh trying to relocate that bitmap's memory. On the other hand, the pixmap automatically created for you by `NewGWorld()` is relocatable. This means that it might get moved around the heap during compaction. Thus, before drawing to an offscreen graphics world, you need to lock the pixmap to prevent it from moving around. After you have finished drawing to the pixmap, unlock it so that it can be moved if necessary during a heap compaction. The following toolbox calls allow you to lock and unlock a graphics world's pixel map. Supply the `portPixMap` field of the graphics world pointer to these calls (we'll see an example shortly).

```
// Locks a movable pixel map in memory.  Call this before
// drawing to an offscreen graphics world.  This function
// returns false if there's no offscreen pixmap to draw to.
Boolean LockPixels( PixMapHandle hPixMap );

// Unlocks a locked pixel map.  Call this after you have
// finished drawing to the graphics world.
void UnlockPixels( PixMapHandle hPixMap );
```

Destroying an Offscreen Graphics World

After you have finished with an offscreen graphics world, simply call the following toolbox routine to free up all resources taken by the graphics world. It certainly beats manually freeing up all the memory associated with an offscreen graphics port and bitmap.

```
// Frees up all memory structure associated with the supplied
// graphics world.
void DisposeGWorld( GWorldPtr pGWorld );
```

Offscreen Graphics World Example

Now that we have seen how to use graphics worlds, let's look at the entire `ConstructScoresWindowOffscreen()` function found in **Scores Window.c**. Notice that the code to create the offscreen graphics world takes only two lines, as opposed to the eight lines needed to set up an offscreen graphics port and bitmap. Actually, a line used to create the offscreen graphics world simply sets up the bitmap variable so that when the offscreen bitmap needs to be copied to the scores window on screen, the graphics world pointer doesn't need to be referenced. In other words, the bitmap variable can be specified regardless of whether that bitmap was manually created or automatically created as part of the graphics world. We'll see an example of this later on in the section on bitmap operations.

```
static GWorldPtr pGWorldScores = nil;
static GrafPtr   pGrafScores = nil;
static BitMap    bitmapScores;
static Rect      rectScores;

void ConstructScoresWindowOffscreen( void )
{
  GrafPtr  pGrafCurrent;
  short    sBitmapRowBytes;
  Size     sizeBitmap;
  short    sPixelDepth;

  // Save the current graphics port so that we can restore it after creating
  // the offscreen graphics port.
  GetPort( &pGrafCurrent );

  // If we are using color graphics (which also means we have 32-bit color
  // QuickDraw), create an offscreen graphics world.  UsingColorGraphics()
  // also returns the number of colors the monitor is set to (the monitor's
  // pixel depth).
  if ( UsingColorGraphics( &sPixelDepth ) )
  {
    // Create the offscreen graphics world, specifying the monitor's pixel
    // depth as the pixel depth of the graphics world.  This will speed up
    // drawing to the screen since no pixel depth conversion will be needed.
    NewGWorld( &pGWorldScores, sPixelDepth, &rectScores, nil, nil, 0 );
```

```
    // Set the high scores window bitmap to the appropriate field of the
    // offscreen graphics world.  This is done so that when we copy the
    // offscreen bitmap to the high scores window onscreen, we don't have to
    // care whether the bitmap was created manually (by executing the code
    // below), or automatically with the graphics world.
    bitmapScores = ((GrafPtr) pGWorldScores)->portBits;
  }

  // Black-and-white support only.  Create the graphics port and bitmap the
  // old fashioned way.
  else
  {
    // Allocate storage for the graphics port record.
    pGrafScores = (GrafPtr) NewPtr( sizeof( GrafPort ) );

    // Initialize (open) the port for the first time.  OpenPort() also sets
    // the given port to be the current graphics port.
    OpenPort( pGrafScores );

    // Compute the number of bytes needed for a single row of the bitmap we
    // plan to associate with the offscreen graphics port.  Remember, the rows
    // need to be word aligned, so round up to the nearest word.
    sBitmapRowBytes = (((rectScores.right-rectScores.left-1) / 16) + 1) * 2;

    // Compute the number of bytes needed for the entire bitmap, which is the
    // number of bytes needed per row times the number of rows.
    sizeBitmap = ( rectScores.bottom - rectScores.top ) * sBitmapRowBytes;

    // Allocate the storage needed for the in-memory bitmap, and set the
    // bitmap fields accordingly.
    bitmapScores.baseAddr = (QDPtr) NewPtr( sizeBitmap );
    bitmapScores.rowBytes = sBitmapRowBytes;
    bitmapScores.bounds = rectScores;

    // Associate the new bitmap with the new graphics port.
    SetPortBits( &bitmapScores );
  }

  // Set the current graphics port back to what it was before this function
  // was called.
  SetPort( pGrafCurrent );
}
```

The following function from **Scores Window.c** frees up the memory associated with the offscreen drawing environment used for the high scores screen.

```
void DestructScoresWindowOffscreen( void )
{
  // If a graphics world was allocated, dispose of it, freeing its resource.
  if ( pGWorldScores )
    DisposeGWorld( pGWorldScores );

  // If a manually created graphics port (and bitmap) was allocated, free
  // all of its resources manually.
  if ( pGrafScores )
  {
    DisposPtr( (Ptr) bitmapScores.baseAddr );
    ClosePort( pGrafScores );
    DisposPtr( (Ptr) pGrafScores );
  }

  // Invalidate the pointers, so that we know that no offscreen graphics
  // world or port exists for the high scores window.
  pGWorldScores = nil;
  pGrafScores = nil;
}
```

Determining the Macintosh's Graphics Environment

How do you tell if the Macintosh your game is running on supports 32-bit color QuickDraw? In addition, how do you tell if the user has the monitor set to black and white or color? And, even if the monitor is set to **color** mode, how can you tell what the monitor's pixel depth is set to? What happens if the user changes the monitor's pixel depth in the middle of your game? This section will answer these questions.

32-bit Color QuickDraw or Not

First, you need to determine if the Macintosh your game is running on supports 32-bit color QuickDraw. If not, you can't use offscreen graphics worlds. In that case, you have one of three options. First, you could simply tell the user that your game requires 32-bit color QuickDraw and not allow them to play your game. Second, you could manually create all the offscreen color graphics ports and pixmaps yourself. Last, you could do what Desert Trek does and just not support color on those machines; they would have to play the game in black and white.

The following routine from **Offscreen Graphics.c** determines whether or not a Macintosh supports 32-bit color QuickDraw. A portion of the function not related to determining 32-bit color QuickDraw has been left out. Also, I'm not going to describe the `Gestalt()` toolbox call other than to tell you that it is used to obtain the version of QuickDraw used on the computer your game is running on.

```
static Boolean bHas32BitQuickdraw;

static void Check32BitQuickdraw( void )
{
  OSErr  osErr;
  long   lQDVersion = 0;

  // Default to false.
  bHas32BitQuickdraw = false;

  // Check to see if the Gestalt() toolbox function is supported on this
  // Macintosh.
  if ( NGetTrapAddress( _Gestalt, ToolTrap ) !=
       NGetTrapAddress( _Unimplemented, ToolTrap ) )
  {
    // Use the Gestalt() toolbox call to get the QuickDraw version number.
    osErr = Gestalt( gestaltQuickdrawVersion, &lQDVersion );

    // If the version number is greater than or equal to 0x0200, this
    // Macintosh supports 32-bit color QuickDraw.
    if ( lQDVersion >= 0x0200 )
      bHas32BitQuickdraw = true;
  }
}
```

Determining the Monitor's Pixel Depth

The following routine from **Offscreen Graphics.c** determines the monitor's current pixel depth. It uses the toolbox call `GetGDevice()` to obtain the graphics device handle for the screen. A field of the graphics device handle contains the monitor's current pixel depth. You shouldn't need to use graphics devices that are beyond what's covered in the following function:

```
static short GetMonitorPixelDepth( void )
{
  GDHandle  hGDMonitor = nil;
  short     sPixelDepth = 0;

  // Get a handle to the graphics device.
  hGDMonitor = GetGDevice();

  // If we got a valid handle, obtain the monitor's pixel depth.
  if ( hGDMonitor )
    sPixelDepth = (*(*hGDMonitor)->gdPMap)->pixelSize;

  // Return the monitor's pixel depth to the calling routine.
  return( sPixelDepth );
}
```

The monitor's pixel depth will be 1 when the monitor is set to black-and-white mode.

Reacting to Changes in the Monitor's Pixel Depth

To be honest, many games do not take action when the monitor's pixel depth changes. However, if the user changes the monitor's pixel depth in the middle of your game, drawing speed may significantly degrade, or garbled results may appear on the screen. Why would this happen? First, if your offscreen bitmaps have a pixel depth different from the monitor's pixel depth, QuickDraw will need to convert between the two different depths when you copy the offscreen bitmap to the screen. That conversion takes time, and thus the drawing speed of your game will be degraded. We'll discuss bitmap drawing speed in much more detail when

we learn about the `CopyBits()` toolbox routine later. Second, if the conversion goes from a higher pixel to a lower depth, your bitmap might look garbled when it gets displayed on the screen. For example, if you try to draw a beautiful offscreen 256-color picture on a monitor in black-and-white mode, it probably isn't going to look so great.

So, how should a game react to a change in the monitor's pixel depth? First, you need to decide if your game is going to support all monitor pixel depths. If not, you'll probably just display a message to the user and quit. However, if your game will support a pixel depth change, you'll need to reload your game's graphics and rebuild all your offscreen bitmaps to match their depths with the new depth of the monitor. Desert Trek will support all monitor pixel depths, but will use color graphics only if the monitor is set to at least 16 colors. Anything less will cause Desert Trek to draw all its graphics in black and white.

You need to be careful if you do support different pixel depths. Why? Well, the larger the pixel depth, the more memory the offscreen bitmaps will need. You need to make sure there's enough memory available to your game to allocate all the offscreen graphics bitmaps. If not, you will have to create all your offscreen bitmaps at a lower pixel depth, one that can be supported given the amount of memory available to your game. If there isn't enough memory to allocate the offscreen bitmaps at the monitor's pixel depth, Desert Trek will create offscreen bitmaps with the highest pixel depth allowed by the memory available.

When should you check to see if the monitor's pixel depth has changed? That's easy. Whenever your game gets an update event, check the monitor's pixel depth. If the user does change the monitor's pixel depth, all windows on that Macintosh receive an update event so that they can adapt to the new pixel depth. Go back and check out the `HandleUpdateEvent()` function shown in Chapter 4 on working with windows. The first thing it does is call the `CheckMonitorColors()` function to see if the monitor's pixel depth has changed. Here's the code that checks and reacts to changes in the monitor's pixel depth. The function, as well as all the support functions called by `CheckMonitorColors()` not shown here, can be found in **Offscreen Graphics.c**.

```
// Set to true if this Macintosh supports 32-bit color QuickDraw.
static Boolean bHas32BitQuickdraw;

// Set to true if color graphics are being used.  The Macintosh must support
// 32-bit color QuickDraw, and the monitor must be set to a pixel depth of at
// least 4 for this to be true.
static Boolean bUsingColorGraphics;

// Contains the current pixel depth used by all the offscreen bitmaps.
static short   sOffscreenPixelDepth = 0;

// Contains the current pixel depth of the monitor.
static short   sMonitorPixelDepth = 0;

void CheckMonitorColors( Boolean bReloadGraphics )
{
  short      sCurrentPixelDepth;
  short      sNewOffscreenPixelDepth;
  long       lAppSize;
  short      sMaxDepthAllowed;
  short      sMemNeededForDepth;

  // We only care about changing monitor depths if the Mac supports 32-bit
  // color QuickDraw.  Otherwise, graphics for Desert Trek are always drawn
  // in black and white.
  if ( bHas32BitQuickdraw )
  {
    // Get the monitor's current pixel depth.
    sCurrentPixelDepth = GetMonitorPixelDepth();

    // If the pixel depth has changed from the last time we checked, we have
    // some work to do.
    if ( sCurrentPixelDepth != sMonitorPixelDepth )
    {
      // Set the variable that keeps track of the monitor's pixel depth so we
      // can check it again at the next window update.
      sMonitorPixelDepth = sCurrentPixelDepth;

      // Call a function that reads the 'SIZE' resource to see how much memory
      // the user has allocated to Desert Trek.  We will use this number to
      // determine if there's enough memory to allocate all offscreen bitmaps
      // at the monitor's pixel depth.
      lAppSize = GetCurrentAppSize();
```

```
// Compute the maximum offscreen pixel depth allowed given the 'SIZE'
// resource's memory setting.
sMaxDepthAllowed = GetMaxDepthAllowed( lAppSize );

// Compute the amount of memory needed to support the monitor's current
// pixel depth.  This will be used to display a message to the user,
// telling them what they need to set Desert Trek's memory setting to
// in the Get Info box from the Finder in order to support the current
// monitor's pixel depth.
sMemNeededForDepth = GetMemNeededForDepth( sCurrentPixelDepth );

// If the current monitor's pixel depth is greater than what's allowed
// based on the memory available to Desert Trek, we need to scale back
// the offscreen pixel depths to the maximum allowed given the amount of
// memory available.  If the user has color depth warnings turned on,
// display a message explaining that they may see graphics performance
// degradation since there isn't enough memory to support the current
// monitor's pixel depth.
if ( sMonitorPixelDepth > sMaxDepthAllowed )
{
  sNewOffscreenPixelDepth = sMaxDepthAllowed;

  if ( **hbColorDepthWarning )
    DisplayMemoryAlert( sMaxDepthAllowed, sMemNeededForDepth );
}

// Otherwise, there's enough memory.  Set the offscreen pixel depth for
// bitmaps to match that of the screen for best graphics performance.
else
  sNewOffscreenPixelDepth = sMonitorPixelDepth;

// If the offscreen pixel depth has changed, we need to close all
// offscreen graphics ports and worlds, and recreate them at the new
// pixel depth.  The ReloadGraphics() function accomplishes this.  Also
// set the variables that define what graphics mode Desert Trek is
// currently using (color or black and white, and if color, the pixel
// depth used).
if ( sNewOffscreenPixelDepth != sOffscreenPixelDepth )
{
  sOffscreenPixelDepth = sNewOffscreenPixelDepth;
  bUsingColorGraphics = (Boolean) ( sOffscreenPixelDepth >= 4 );
```

```
            if ( bReloadGraphics )
              ReloadGraphics();
          }
        }
      }
    }
```

Drawing Graphics

Now that we have an understanding of where you can draw graphics (whether it is into a window onscreen or a bitmap offscreen), let's look at how you actually draw the graphics. Drawing is divided into two basic procedures: setting up the drawing environment, and performing the actual drawing. Setting up the drawing environment affects how the actual drawing takes place and includes characteristics such as transfer modes, pens, patterns, and colors. Once you've set the characteristics, you can draw lines, rectangles, rounded rectangles, ovals, pictures, and icons (there are also drawing routines for polygons, arcs, and regions, but they are less frequently used and won't be covered in this book). This section will discuss specific drawing commands that you can issue to affect bitmaps and windows (when I use the term bitmap for the rest of this section, I mean an offscreen bitmap or the content region of an onscreen window). Drawing text and transferring images from one bitmap to another will be covered in later sections.

Keep in mind that all drawing routines discussed here affect the current graphics port or graphics world (and you can assume that when I say graphics port in the future, I also mean graphics world). Make sure you set the current graphics port to the port you wish to draw to before using these routines. Remember, you can draw directly to a window onscreen by setting the current graphics port to the window pointer of the window you wish to draw to. However, most of the time you will perform these drawing commands to offscreen bitmaps and then transfer the completed images to windows and dialog boxes onscreen using bitmap transfer operations.

Patterns

Patterns are used when filling areas of a bitmap or when drawing lines and other objects into a bitmap. A standard black-and-white pattern is an 8 bit by 8 bit structure that determines how an area of the bitmap gets filled, and repeats itself over the entire affected area (see Figure 7.2). Color patterns are a little more complex, so we'll skip showing the actual data structure. Think of them as an 8 pixel by 8 pixel pattern where each pixel can be any color as opposed to just black and white. Don't worry about their additional complexity, though, because color patterns are just as easy to use as standard patterns.

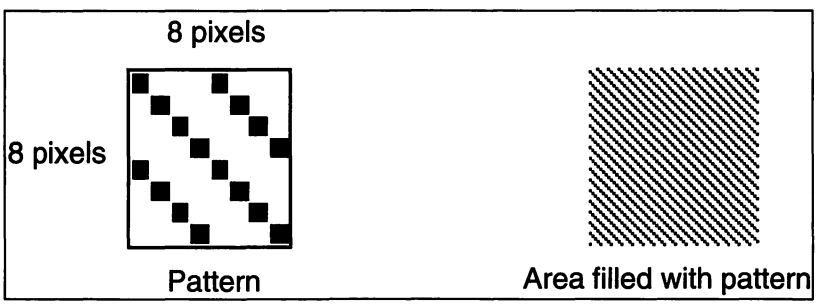


Figure 7.2 How patterns work.

```
struct Pattern{
  unsigned char pat[8];
};

typedef struct Pattern Pattern;
typedef Pattern *PatPtr;
typedef PatPtr *PatHandle;

// Color pattern structure, PixPat.
typedef struct PixPat PixPat;
typedef PixPat *PixPatPtr, **PixPatHandle;
```

Most of the time, you will use a black pattern when drawing, which means that a solid line or object will be drawn. However, you may wish to

use different patterns to draw certain special effects. Figure 7.3 shows how different patterns affect the drawing of a thick line, or the filling of a rectangle (the toolbox calls used to draw these objects will be described shortly). Desert Trek uses patterns when drawing the view transition from one part of the day to the next. Later in this chapter, we'll see how to use patterns to provide this effect.

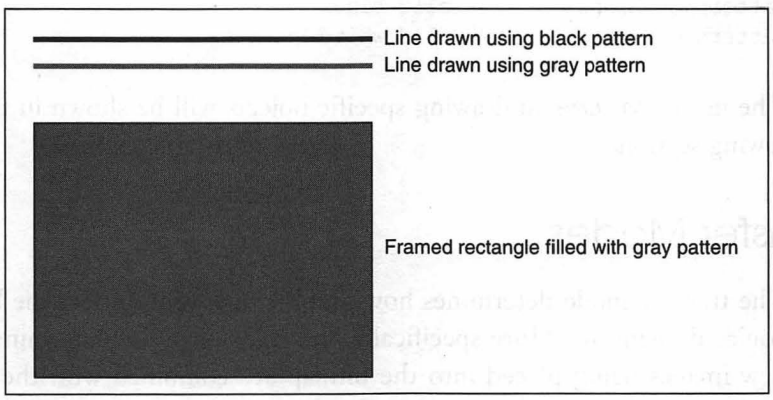


Figure 7.3 How patterns affect the drawing of a line.

Patterns can be loaded from the resource fork using either the `GetPattern()` or `GetIndPattern()` toolbox calls. Color patterns, often referred to as *pixel* patterns, can be loaded using the `GetPixPat()` toolbox call.

```
// Reads a pattern of type 'PAT ' from the resource fork.
PatHandle GetPattern( short sPatternID );

// Reads a pattern of type 'PAT#' from the resource fork.
void GetIndPattern( Pattern     pattern,
                    short       sPatternListID,
                    short       sPatternIndex );

// Reads a color pixel pattern of type 'ppat' from the
// resource fork.
PixPatHandle GetPixPat( short sPixPatID );
```

The following standard patterns are built into the toolbox and don't need to be loaded from a pattern resource. You can use them directly in any toolbox call that takes a pattern as a parameter.

```
Pattern qd.dkGray;    // Dark gray
Pattern qd.ltGray;    // Light gray
Pattern qd.gray;      // Medium gray
Pattern qd.black;     // Solid black
Pattern qd.white;     // Solid white
```

The use of patterns in drawing specific objects will be shown in the following sections.

Transfer Modes

The transfer mode determines how graphic operations affect the bitmap you're drawing to. More specifically, the transfer mode determines how new images being placed into the bitmap are combined with the image already contained in that bitmap. When I say new image, I really mean anything drawn to the bitmap (lines, rectangles, text, or portions of other bitmaps). Most of the time you will want the new image to replace what's already contained in the bitmap you're drawing to, but there may be times when you want the new image combined in some way with the existing image. There are two classes of transfer modes defined by the Macintosh toolbox, one that affects the drawing of lines, shapes, and patterns, and another that affects bitmap transfer operations and the drawing of text. The following transfer modes are defined by the toolbox.

```
enum {
     // Transfer modes for bitmap transfer operations and
     // the drawing of text.
     srcCopy = 0,
     srcOr = 1,
     srcXor = 2,
     srcBic = 3,
     notSrcCopy = 4,
     notSrcOr = 5,
     notSrcXor = 6,
```

```
        notSrcBic = 7,

        // Transfer modes for pens and patterns.
        patCopy = 8,
        patOr = 9,
        patXor = 10,
        patBic = 11,
        notPatCopy = 12,
        notPatOr = 13,
        notPatXor = 14,
        notPatBic = 15
}
```

Individual elements of each group affect graphics operation in the same way, meaning that `srcCopy` does the same thing for bitmaps and text that `patCopy` does for pens and patterns.

When the new image is placed over an existing image, the bits of both images are combined using the method specified by the transfer mode. The following tables show how the bits are combined. Note that even though the table shows how black-and-white bits are combined, the same rules apply for color pixels. Color pixels are represented by more than 1 bit, and those individual bits are combined using the same rules defined for the combination of black-and-white pixels (which are represented by a single bit). For example, a 16-color image contains 4 bits of color information for each pixel. A 256-color image requires 8 bits for each pixel. For black-and-white images, white pixels are considered to have a bit value of 0 and black pixels a bit value of 1. We'll learn more about color pixels' bit representation later.

srcCopy and *patCopy*

These transfer modes replace existing pixels with pixels from the new image. Most of the time, you will use these transfer modes because you'll want what you're drawing to replace the existing image on the bitmap you're drawing to. The following table shows how pixels from the new image are combined with the existing image. In this case, no real combination takes place because the existing pixels are simply replaced with the new pixels.

New Pixel	Existing Pixel	Resultant Pixel
white	white	white
white	black	white
black	white	black
black	black	black

srcOr and *patOr*

These transfer modes combine the pixels of both the new and existing image using the logical or operation. Visually, the new image overlays the existing image. If a pixel is "on" in either the new or existing image, the pixel will be "on" in the resultant image. A pixel is considered "on" when its bit is set to **1**. For a black-and-white image, this means a black pixel.

New Pixel	Existing Pixel	Resultant Pixel
white	white	white
white	black	black
black	white	black
black	black	black

srcXor and *patXor*

These transfer modes combine the pixels of both the new and existing images using the logical exclusive-or operation. The result is that selected pixels in the existing image are inverted. The selection is determined by the new image. Black pixels in the new image cause the corresponding pixels in the existing image to be inverted. White pixels in the new image have no effect on the corresponding pixels in the existing image. For example, you could invert all the pixels in a bitmap by setting the transfer mode to **patXor**, and drawing a black-filled rectangle the size of that bitmap (we'll see a better way to invert images later in this chapter).

New Pixel	Existing Pixel	Resultant Pixel
white	white	white
white	black	black
black	white	black
black	black	white

srcBic and *patBic*

These transfer modes cause selected pixels in the existing image to be reset to **white**. The selection is determined by the new image. Black pixels in the new image cause the corresponding pixels in the existing image to be reset to **white**. White pixels in the new image have no effect on the corresponding pixels in the existing image.

New Pixel	Existing Pixel	Resultant Pixel
white	white	white
white	black	black
black	white	white
black	black	white

The `notSrc` and `notPat` transfer modes basically reverse the effect black and white pixels in the new image have on the existing image.

notSrcCopy and *notPatCopy*

The resultant image is an inverted copy of the new image.

New Pixel	Existing Pixel	Resultant Pixel
white	white	black
white	black	black
black	white	white
black	black	white

notSrcOr and *notPatOr*

Black pixels in the new image have no effect on the existing image, and white pixels in the new image result in black pixels regardless of what existed previously.

New Pixel	Existing Pixel	Resultant Pixel
white	white	black
white	black	black
black	white	white
black	black	black

notSrcXor and *notPatXor*

White pixels in the new image cause pixels in the existing image to be inverted. Black pixels in the image have no effect on those in the existing image.

New Pixel	Existing Pixel	Resultant Pixel
white	white	black
white	black	white
black	white	white
black	black	black

notSrcBic and *notPatBic*

These transfer modes combine the pixels of both the new and existing images using the logical and operation. In other words, pixels in the resulting image are black only if pixels in the new image and existing image are black.

New Pixel	Existing Pixel	Resultant Pixel
white	white	white
white	black	white
black	white	white
black	black	black

Pens

The pen is used to draw lines as well as to outline objects such as rectangles and ovals. A bitmap's pen has several characteristics that affect images drawn using the pen. These characteristics include pen position, size, mode, and pattern. The pen's current position determines where pen drawing operations start. This means that any line or text drawn will start at the pen's current location. You will frequently move the pen to the desired start location just before drawing lines and text . The pen's size determines the width and height of lines and object outlines (such as the outlines of rectangles of ovals). The pen's mode is the transfer mode used when drawing objects with the pen. Though text location is determined by the pen's location, the transfer mode used for text is not determined by the pen's mode. Later in this chapter, we'll see that the text mode is specified separately. Finally, the pen's pattern is used when drawing objects with the pen. For example, if you draw a line when the pen's pattern is set to **gray**, you get a gray line. If the pattern is set to **white**, you get a white line. The following record contains the pen's characteristics.

```
struct PenState {
     Point     pnLoc;    // Pen's location
     Point     pnSize;   // Pen's size
     short     pnMode;   // Pen's transfer mode
     Pattern   pnPat;    // Pen's pattern
};
```

```
typedef struct PenState PenState;
```

You can query and set the pen's state record using the following toolbox calls:

```
// Reads the pen's current state record into the supplied
// variable.
void GetPenState( PenState *ppenState );

// Set's the pen's state to the specified state record.
void SetPenState( PenState penState );
```

Usually, you will only need to change one of the pen's characteristics at a time. For example, you may want to move the pen's location to the start of a line, or to where you would like to draw some text. It doesn't really make sense to make a call to `SetPenState()` every time you want to change only one field of the pen's state record. For that reason, the toolbox provides the following calls, which affect one record of the pen's state at a time. Keep in mind that all these calls affect the pen for the current graphics port. Make sure you've set the current graphics port to the graphics port you want to affect.

```
// Get the pen's current location.
void GetPen( Point point );

// Move the pen to the specified location.
void MoveTo(   short     sHPosition,
               short     sVPosition );

// Move the pen to by the specified offsets.  Positive values
// move the pen down and to the right, negative values move
// the pen up and to the left.
void Move(     short     sHOffset,
               short     sVOffset );

// Set the pen's width and height.
void PenSize( short      sWidth,
               short     sHeight );

// Set the pen's pattern.
void PenPat( Pattern pattern );
```

```
// Set the pen's transfer mode.  Use one of the pattern modes
// defined above.
void PenMode( short sMode );

// Set all pen characteristics to their defaults.  This means
// a width and height of 1, a pattern of solid black, and a
// transfer mode of patCopy.
void PenNormal( void );
```

Last, you can show and hide the pen. A hidden pen does not drawing anything in its graphics port. In other words, lines, object outlines, and text do not get drawn if the pen is hidden, even if you issue drawing commands. The following toolbox routines show and hide the pen.

```
// Hides the pen so that no drawing operations take place.
void HidePen( void );

// Shows the pen so that drawing operations take place.
void ShowPen( void );
```

Color

An entire book could easily be devoted to explaining the various aspects of how the Macintosh supports color graphics. However, the basics needed to write games can be covered in a few paragraphs.

Most everything discussed in this chapter so far has centered around black-and-white graphics. At first that might seem odd because most games written today need to support color graphics if they are to be successful. Why spend so much time on black-and-white graphics and just mention color almost as a side note? The answer is that just about everything that applies to black-and-white graphics applies equally to color graphics. Adding color to your game is very simple once you understand the fundamentals of drawing. This section will show you how easy it is to draw in color.

When drawing in black and white, a pixel is considered "on" when that pixel is black and "off" when that pixel is white. Thus a single pixel in a black-and-white bitmap requires only 1 bit; where a white pixel has a

bit value of 0, a black pixel has a bit value of 1. When drawing in color, a single pixel can be more that just black and white. It can be red, green, blue, black, white, or just about any other color. This is accomplished by providing more than 1 bit per pixel. A single bit can represent two values, or two colors, which are typically black and white. To represent more colors, you need more bits. The number of bits allocated for each pixel determines how many colors can be supported in a graphics port. The following table shows what color depths the Macintosh supports.

Bits per Pixel	Number of Colors
1	2 (black and white)
2	4
4	16
8	256
16	65,536
24	16.7 million

When you draw in color to a graphics port, you need to specify the color of the object to be drawn. The color you want to draw is typically specified by providing the red, green, and blue components of the color you want to draw. This is commonly referred to as an RGB value, and the toolbox defines an RGB structure used to specify color.

```
struct RGBColor {
     unsigned short red;      // Red component
     unsigned short green;    // Green component
     unsigned short blue;     // Blue component
};

typedef struct RGBColor RGBColor;
```

When you specify an RGB color to draw, that color most likely will need to be converted into a color supported by that graphics port. From the preceding RGB definition, you can obviously specify a lot more colors than a graphics port might support. For example, you can certainly spec-

ify more RGB colors than a 256-color graphics port will be able to show. How does an RGB color specification get translated into a specific color in the graphics port? When you specify an RGB color, the Macintosh toolbox automatically chooses the color supported by the graphics port that is closest to the color you specify. This leads to another question. What colors does a graphics port of a specific color depth support? The Macintosh uses a color table for each graphics port to list the colors supported by that graphics port. When you create a graphics port, you can specify the color table used. The default color table will be used if you choose not to supply your own. In most cases, unless your game has very specific color needs not taken care of by the default color table, you should use the default color table for your game's graphics. If you change the Macintosh screen's color table, all graphics shown on the screen will be drawn using your own color table, causing other application's windows to use your colors. Other applications, which include the Macintosh Finder, might not look aesthetically pleasing in your colors.

The following table shows the RGB values used for a standard 4-bit, 16-color graphics port.

Color	Red	Green	Blue
Black	0	0	0
Dark Gray	16448	16448	16448
Medium Gray	32869	32869	32869
Light Gray	49344	49344	49344
Tan	37008	29041	14906
Brown	22102	11308	1285
Green	0	25700	4369
Light Green	7967	47031	5140
Light Blue	514	43947	60138
Blue	0	0	54484
Purple	17990	0	42405
Pink	62194	2056	33924

Red	56797	2056	1542
Orange	65535	25700	514
Yellow	64764	62451	1285
White	65535	65535	65535

How do you specify an RGB color, and what effect will that have on drawing? Again, let's recall how black-and-white drawing takes place. When you draw in black and white, your drawing will either cause certain pixels to turn white, black, or remain the same. For example, if you draw a line with the standard pen (solid black pattern and a mode of `patCopy`), a solid black line will be drawn to the graphics port, causing all pixels the line affects to turn black. If you fill a rectangle with a gray pattern with the transfer mode set to **patCopy**, every other pixel in the rectangle will be changed to black, the others changed to white (refer back to Figure 7.3).

When drawing in color, the graphics port's foreground color determines what color to draw pixels that would have been drawn black in a black-and-white graphics port. The port's background color determines what color to draw pixels which would have been drawn white in a black-and-white port. For example, if a port's foreground color is red, lines drawn will appear in red (assuming, of course, a pen transfer mode of `patCopy`). Filling a rectangle with a gray pattern will cause half the pixels to be of the foreground color, the other half to be of the background color. For example, if the port's foreground color is red and the background color is green, a rectangle filled with a gray pattern will have half its pixels red, the other half green (the red corresponding to what would have been black, and the green corresponding to what would have been white in a black-and-white graphics port). This means that drawing in color takes place exactly the same as drawing in black and white, except that you need to set the foreground and background colors before you draw.

To set the current graphics port's foreground and background colors or to determine their current values, use the following toolbox calls.

```
// Set the current graphics port's foreground color.
void RGBForeColor( RGBColor *prgbColor );

// Set the current graphics port's background color.
void RGBBackColor( RGBColor *prgbColor );

// Get the current graphics port's foreground color.
void GetForeColor( RGBColor *prgbColor );

// Get the current graphics port's background color.
void GetBackColor( RGBColor *prgbColor );
```

Most of the time, you will only need to set a graphics port's foreground color when drawing because most drawing commands are used to set pixels, not reset them (set pixels are drawn black, or in the foreground color, reset pixels are drawn white, or in the background color). It is generally a good idea to query a graphics port's foreground and background colors before changing them so that you can set them back when you've finished drawing. This is especially true when drawing to the Macintosh screen or a window that draws some of its own contents, such as controls. Also, before using any bitmap transfer operation, you must set the foreground color to **black** and the background color to **white** before performing that operation. Otherwise, undefined results will occur.

TIP

When you need to specify a color to draw into your graphics port, how can you easily choose an RGB color that matches the color you want? Desert Trek defines an array of 16 RGB colors that are read from a custom resource. These colors correspond to the Macintosh's standard 16 colors, and thus look the same whether the user has the monitor set to **16**, **256**, **thousands**, or **millions** of colors. When Desert Trek want to draw an object in color, it sets the foreground color to one of these values. The following structure definition and enumeration defines the 16-bit RGB color array.

```
enum enumStd16ColorsIndex { ColorWhite = 0, ColorYellow, ColorOrange,
                            ColorRed, ColorPink, ColorPurple, ColorBlue,
                            ColorLightBlue, ColorLightGreen, ColorGreen,
                            ColorBrown, ColorTan, ColorLightGray,
                            ColorMediumGray, ColorDarkGray, ColorBlack
};

typedef struct _Colors
{
  RGBColor  rgbColor[16];
} Colors, *pColors, **hColors;
```

The following line of code loads the colors from Desert Trek's resource file.

```
static hColors hStd16Colors;

hStd16Colors = (hColors) GetResource( 'clrs', 128 );
```

When Desert Treks want to set the foreground color to one of the standard 16 colors, it uses the `hStd16Colors` handle, which points to an array of the RGB values corresponding to the Macintosh's standard 16 colors. The following is an example.

```
// Set the foreground color to yellow.
RGBForeColor( &(*hStd16Colors)->rgbColor[ColorYellow] );
```

If you would like to use this code in your game, make sure to copy the `'clrs'` resource from Desert Trek into your game's resource fork.

Drawing Lines

The pen draws lines in a graphics port. To draw a line, you first need to move the pen to the start point of the line you want to draw. The toolbox call that draws lines does so by moving the pen to a specified end point. Moving the pen from one location to another draws consecutive lines where the start point of one line is the same as the end point of the previous line. The pen's characteristics as described above in the section on

pens affect how the line will be drawn in a graphics port. This includes the line's width, pattern, and transfer modes. Figure 7.4 shows a variety of lines drawn based on different pen settings.

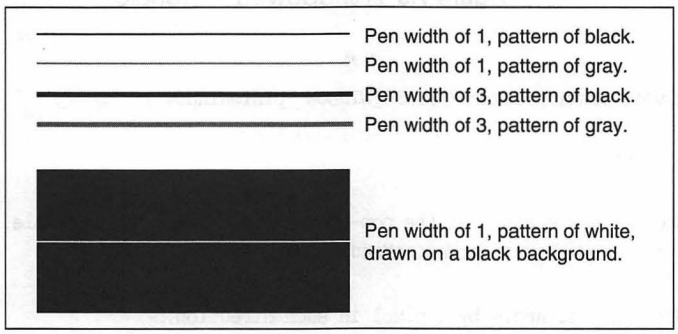


Figure 7.4 Various lines.

The following two toolbox calls draw a line. Note that you specify the end points in the calls. The start point is always the pen's current location.

```
// Draw a line the specified horizontal and vertical distance
// from the pen's current location.
void Line(    short      sHDistance,
              short      sVDistance );

// Draw a line to the specified coordinates from the pen's
// current location.
void LineTo( short       sHCoordinate,
              short      sVCoordinate );
```

Let's look at an example of how to draw lines. Way back in Chapter 5 on menus, it was stated that when using pop-up menus, you are responsible for drawing the area of the screen that signifies where the user should click to pop up a menu. That area is typically a framed rectangle with a shadow (see Figure 7.5). The shadow can be drawn using lines. The following routine from Desert Trek draws the pop-up menu box with a shadow and can be found in **Information Window.h**.

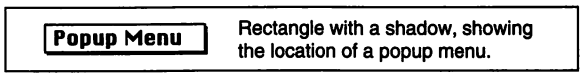


Figure 7.5 A shadowed rectangle.

```
static void DrawTopicsMenu( PINFO_WINDOW  pInfoWindow )
{
  Rect  rect;

  // Store the rectangle of the pop-up menu in a temporary variable.
  rect = pInfoWindow->rectTopicsMenu;

  // Grow the rectangle by 1 pixel in each direction.
  InsetRect( &rect, -1, -1 );

  // Outline the rectangle.  We'll see this call defined in the next section.
  FrameRect( &rect );

  // Move the pen to the bottom left of the pop-up menu location, moving 2
  // pixels to the right to produce a shadow effect.
  MoveTo ( rect.left + 2, rect.bottom );

  // Draw a line to the bottom right of the pop-up menu's location.
  LineTo ( rect.right, rect.bottom );

  // Draw a line from the previous line's end point to 2 pixels below the top
  // right of the pop-up menu's location (again, to produce the shadow effect).
  LineTo ( rect.right, rect.top + 2 );
}
```

Drawing Rectangles

Rectangles form one of the most fundamental shapes drawn. The following toolbox routines allow you to outline, draw, and fill rectangles with a specified pattern. The pen's size, pattern, and transfer mode affect most of the following operations.

```
// Frame a rectangle, which draws an outline around that
// rectangle.  The pen's style will determine the style of
// the outline drawn (including the width and height of the
// outline).
void FrameRect( Rect *pRect );

// Paints a rectangle using the pen's current pattern and
// transfer mode.
void PaintRect( Rect *pRect );

// Fills a rectangle using the specified pattern.  The
// transfer mode is always patCopy.
void FillRect(  Rect     *pRect,
                Pattern  pattern );

// Fills a rectangle using the specified color pixel pattern.
// The transfer mode is always patCopy.
void FillCRect(  Rect          *pRect,
                 PixPatHandle  hPixPat );

// Erases a rectangle, setting it to the background color.
void EraseRect( Rect *pRect );

// Inverts all pixels in a bitmap contained within the
// specified rectangle.
void InvertRect( Rect *pRect );
```

A number of other code examples in this book draw rectangles, so there's no need here to show an additional example.

Drawing Rounded Rectangles

Rounded rectangles are rectangles with rounded corners. A perfect example of a rounded rectangle is any standard Macintosh push button. Rounded rectangles are drawn like normal rectangles, with the addition of two parameters. You need to specify the width and height of the corners. The width and height really specify the diameters of an oval, which is then used to determine how the rounded corners will look. Each

rounded corner will be drawn using one quarter of the oval (see Figure 7.6). The following toolbox routines are used to draw rounded rectangles.

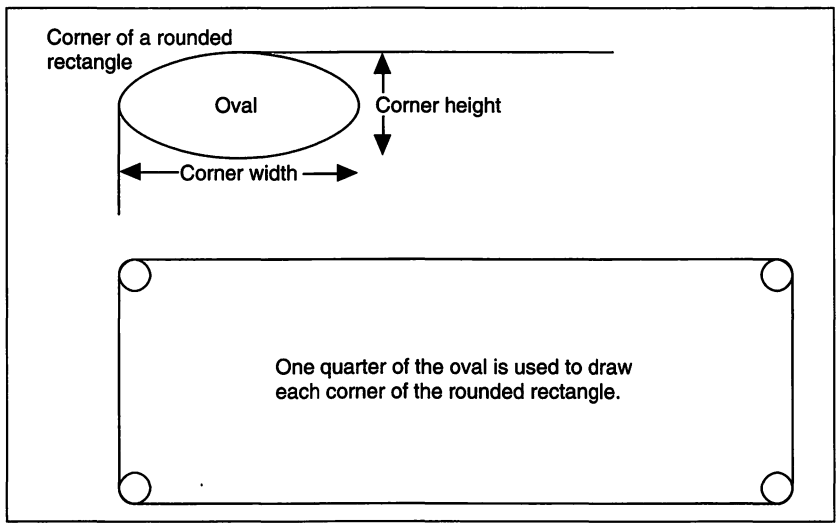


Figure 7.6 The corners of a rounded rectangle.

```
// Frame a rounded rectangle, which draws an outline around
// that rounded rectangle.  The pen's style will determine
// the style of the outline drawn (including the width and
// height of the outline).
void FrameRoundRect(    Rect            *pRect,
                        short           sCornerWidth,
                        short           sCornerHeight );

// Paints a rounded rectangle using the pen's current pattern
// and transfer mode.
void PaintRoundRect(    Rect            *pRect,
                        short           sCornerWidth,
                        short           sCornerHeight );

// Fills a rounded rectangle using the specified pattern.
// The transfer mode is always patCopy.
void FillRoundRect(     Rect            *pRect,
                        short           sCornerWidth,
                        short           sCornerHeight,
                        Pattern         pattern );
```

```
// Fills a rounded rectangle using the specified color pixel
// pattern.  The transfer mode is always patCopy.
void FillCRoundRect(    Rect            *pRect,
                        short           sCornerWidth,
                        short           sCornerHeight,
                        PixPatHandle    hPixPat );

// Erases a rounded rectangle, setting it to the background
// color.
void EraseRoundRect(    Rect            *pRect,
                        short           sCornerWidth,
                        short           sCornerHeight );

// Inverts all pixels in a bitmap contained within the
// specified rounded rectangle.
void InvertRoundRect(   Rect            *pRect );
                        short           sCornerWidth,
                        short           sCornerHeight );
```

Round rectangles are great for showing which push button of a window or dialog box is the default button. Remember, the default button gets "clicked on" when the user presses the **Return** or **Enter** key. The following code example shows how to display to the user which is the default push button. This is accomplished by drawing a thick, rounded rectangle around that push button (see Figure 7.7). This routine comes from **App Modal Dialog.c** and draws all application modal dialog boxes.

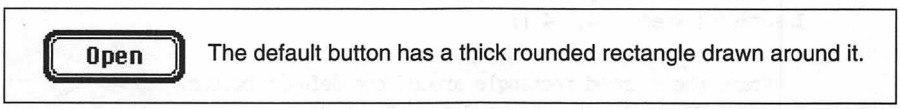


Figure 7.7 A default button.

```
void UpdateModalDialog( DialogPtr pDialog )
{
  GrafPtr  pGrafCurrent;
  Rect     rect;
  short    sCornerSize;
  short    sItem;
  Handle   hItem;
```

```
// Save the current graphics port and change it to the dialog we're about to
// draw.
GetPort( &pGrafCurrent );
SetPort( pDialog );

// Begin updating the dialog.
BeginUpdate( pDialog );

// Call the toolbox routine that draws a dialog and all its items.
DrawDialog( pDialog );

// If there's a default push button, draw the thick outline around that
// button so that the user visually see which button is the default.
if ( pCurrentDialog->sDefaultActionID )
{
  // We need to query the rectangle bounding the default button since it
  // will be used to draw the rounded rectangle around that button.
  GetDItem ( pDialog, pCurrentDialog->sDefaultActionID, &sItem, &hItem,
             &rect );

  // The corner size of rounded rectangle we want to draw around the default
  // button is the height of the button minus 4.
  sCornerSize = rect.bottom - rect.top - 4;

  // The rounded rectangle we want to draw around the default button needs
  // to be 3 pixels thick.  That's accomplished by setting the pen's size.
  PenSize( 3, 3 );

  // The rounded rectangle we want to draw needs to be positioned 4 pixels
  // away from each side of the default button.  So, grow the rectangle of
  // default button by 4 pixels in each direction.
  InsetRect( &rect, -4, -4 );

  // Frame the rounded rectangle around the default button.
  FrameRoundRect( &rect, sCornerSize, sCornerSize );

  // Restore the pen's size.
  PenSize(1, 1);
}

// If there's a custom drawing routine specified for the application modal
// dialog, call it.
CallCustomDrawRoutine( pDialog );
```

```
    // We are finished updating the dialog.
    EndUpdate( pDialog );

    // Restore the current graphics port back to what it was before this routine
    // was called.
    SetPort( pGrafCurrent );
}
```

Drawing Ovals

You draw ovals by specifying the rectangle in which the oval is inscribed (see Figure 7.8). If you want to draw a circle, simply specify a square. The following toolbox routines allow you to draw ovals.

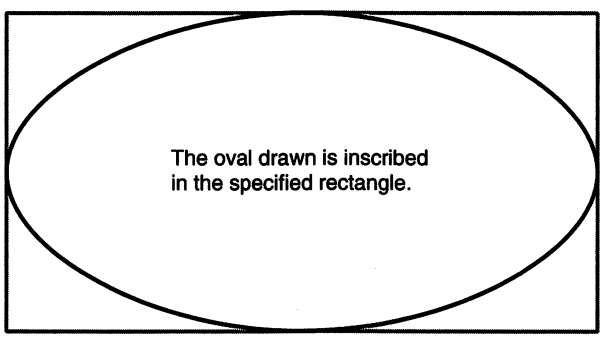


Figure 7.8 An oval.

```
// Frame an oval, which draws an outline around that oval.
// The pen's style will determine the style of the outline
// drawn (including the width and height of the outline).
void FrameOval( Rect *pRect );

// Paints an oval using the pen's current pattern and
// transfer mode.
void PaintOval( Rect *pRect );

// Fills an oval using the specified pattern.  The transfer
// mode is always patCopy.
void FillOval(   Rect            *pRect,
```

```
                Pattern         pattern );

// Fills an oval using the specified color pixel pattern.
// The transfer mode is always patCopy.
void FillCOval( Rect            *pRect,
                PixPatHandle    hPixPat );

// Erases an oval, setting it to the background color.
void EraseOval( Rect *pRect );

// Inverts all pixels in a bitmap contained within the
// specified oval.
void InvertOval( Rect *pRect );
```

Drawing Icons

You can create icons in the resource fork of your game using ResEdit, and later load and draw those icons into a graphics port. Black-and-white icons have a resource type of 'ICON', and color icons have a resource type of 'cicn'. The following toolbox calls load icons from the resource fork.

```
// Loads a black and white icon of type 'ICON' from the
// resource fork.  Use ReleaseResource() when you are
// finished with the icon.
Handle GetIcon( short sIconID);

// Loads a color icon of type 'cicn' from the resource fork.
// Use DisposCIcon when you are finished with the color icon.
CIconHandle GetCIcon( short sIconID );

// Releases the resources taken up by a color icon.
void DisposCIcon( CIconHandle hCIcon );
```

To draw an icon, use one of the following two toolbox calls. Notice that you can stretch an icon to fit inside a given rectangle. Icons are typically 32 pixels wide by 32 pixels high, so if you don't want any stretching to take place, make sure the specified rectangle has a width and height of 32 pixels. Icons are drawn using an *icon mask*. An icon mask is always black and white, even though the icon itself could be color. The black pixels in

the icon mask determine which pixels from the icon get drawn into the destination graphics port. A white pixel in the mask means that the corresponding pixel in the icon does not get drawn. You build the icon's mask when creating a color icon in ResEdit. For black-and-white icons, the icon itself is it's own mask (since both are black and white).

```
// Draws a black and white icon into the current graphics
// port within the specified rectangle.
void PlotIcon(  Rect            *pRect,
                Handle          hIcon );

// Draws a color icon into the current graphics port within
// the specified rectangle.
void PlotCIcon( Rect            *pRect,
                CIconHandle     hCIcon );
```

Drawing Pictures

You can draw pictures using your favorite graphics editor and then place them into the resource fork of your game as type 'PICT'. Your game can then load these pictures and draw them into a graphics port. However, let me give you this one warning: Drawing pictures takes a long time, so you should never draw the same picture multiple times into any of your game windows. Instead, always create an offscreen bitmap to hold your pictures so that they can be copied later into other graphics ports (such as a window, dialog box, or another graphics port) using one of the bitmap transfer operation discussed below. In other words, when your game initializes itself, it should create offscreen bitmaps to hold the pictures loaded from the resource fork. You should then load the pictures, draw them to the offscreen graphics ports, and release the picture resources (you won't need the picture resources anymore because they were already copied to an offscreen bitmap).

Desert Trek creates three distinct offscreen graphics ports to hold three pictures loaded from the resource fork of the game. The pictures loaded include the general game's graphics (such as the 3D picture buttons and objects placed in the view portion of the game window), the various views of

the desert including the jail and trading post, and the game's graphics masks (we'll discuss the use of the masks in the section on bitmap operations).

The following toolbox call loads a picture resource of type 'PICT'.

```
// Loads a picture resource of type 'PICT' from the resource
// fork.  Use ReleaseResource() when you are finished with
// the picture.
PicHandle GetPicture( short sPictureID );
```

The following toolbox call draws a picture into the current graphics port. You need to specify the rectangle in which to draw the given picture. The picture will be stretched to fit the rectangle specified.

```
// Draw a picture into the specified rectangle.
void DrawPicture(  PicHandle   hPicture,
                   Rect        *pRect );
```


TIP

You will rarely want a picture loaded from your resource fork to be stretched. So how do you know the size of a picture? You could probably figure it out when you draw the picture in your favorite graphics editor, but surely there's a better way. Pictures are in fact fairly complex graphical objects. They are described by a large record, which contains the many properties associated with that picture. One of the fields in a picture's record is its bounding rectangle. The field name is `picFrame`. Thus, when you load a picture, you can determine its size by looking at the `picFrame` field of the picture record. You might want to use this information to determine the size of the bitmap needed to hold that picture. Desert Trek uses this field for just such a purpose. The following code creates an offscreen graphics world or a black-and-white graphics port to contain Desert Trek's main graphic elements (the 3D buttons, status gauges, and objects that appear in the view portion of the game window such as the oasis, caravan trading post, etc.). Notice that the picture containing the graphics is loaded from the resource fork first so that the size of the bitmap needed to contain the graphics can be determined before actually creating it. This code can be found in **Offscreen Graphics.c**.

```
static void CreateGraphicsOffscreen( void )
{
  CGrafPtr   pCGrafCurrent;
  GDHandle   hGDCurrent;
  GrafPtr    pGrafCurrent;
  short      sBitmapRowBytes;
  Size       sizeBitmap;
  Rect       rectGraphics;
  PicHandle  hPicGraphics;
  short      sPictureID;

  // Save the current graphics port.
  GetPort( &pGrafCurrent );

  // If we are using color graphics, a new graphics world will be created.
  // Save the current graphics world so that we can restore it at the end
  // of this routine.  Also, we want to load the color picture.
  if ( bUsingColorGraphics )
  {
    sPictureID = COLOR_GRAPHICS_ID;
    GetGWorld( &pCGrafCurrent, &hGDCurrent );
  }

  // Black and white graphics are being used, so load the black and white
  // picture from the resource fork.
  else
    sPictureID = BW_GRAPHICS_ID;

  // Load the picture.
  hPicGraphics = GetPicture( sPictureID );

  // Lock the picture's handle while we're accessing the fields of the
  // picture handle.
  HLock( (Handle) hPicGraphics );

  // Get the picture's bounding rectangle.
  rectGraphics = (*hPicGraphics)->picFrame;

  // In case the picture's bounding rectangle isn't zero based (in other
  // words, the left and top don't have coordinates of 0), offset the
  // rectangle so that the left edge and top edge have a coordinate of 0.
  OffsetRect( &rectGraphics, -rectGraphics.left, -rectGraphics.top );

  // If we're using color graphics, create an offscreen graphics world.
```

```
if ( bUsingColorGraphics )
{
  // Create the offscreen graphics world.
  NewGWorld( &pGWorldGraphics, sOffscreenPixelDepth, &rectGraphics,
             nil, nil, 0 );

  // Set the current graphics world to the newly create graphics world.
  SetGWorld( pGWorldGraphics, nil );

  // Lock the pixel map of the graphics world before drawing to it.
  if ( LockPixels( pGWorldGraphics->portPixMap ) )
  {
    // Erase the offscreen graphics world.
    EraseRect( &rectGraphics );

    // Draw the picture loaded from the resource fork into the graphics
    // world.
    DrawPicture( hPicGraphics, &rectGraphics );

    // We're finished drawing, so unlock the pixel map so it can float in
    // memory as needed (during heap compaction).
    UnlockPixels( pGWorldGraphics->portPixMap );
  }

  // Set the bitmap pointing to the offscreen graphics to be the pixel map
  // of the offscreen graphics world.  It is this bitmap that is referenced
  // whenever copying these graphics to another offscreen bitmap or onscreen
  // window or dialog.
  bitmapGraphics = ((GrafPtr) pGWorldGraphics)->portBits;
}

// If we're using black and white graphics, create an offscreen graphics
// port and corresponding bitmap.
else
{
  // Allocate storage for the graphics port and initialize the port.
  pGrafGraphics = (GrafPtr) NewPtr( sizeof ( GrafPort ) );
  OpenPort( pGrafGraphics );

  // Determine the size of the bitmap needed.
  sBitmapRowBytes = (((rectGraphics.right-rectGraphics.left-1)/16)+1)*2;
  sizeBitmap = ( rectGraphics.bottom - rectGraphics.top ) * sBitmapRowBytes;

  // Allocate storage for the offscreen bitmap to be associated with the
```

```
    // offscreen graphics port.
    bitmapGraphics.baseAddr = (QDPtr) NewPtr( sizeBitmap );
    bitmapGraphics.rowBytes = sBitmapRowBytes;
    bitmapGraphics.bounds = rectGraphics;

    // Associate the offscreen bitmap with the offscreen graphics port.
    SetPortBits( &bitmapGraphics );

    // Erase the offscreen graphics port.
    EraseRect( &rectGraphics );

    // Draw the picture loaded from the resource fork into the offscreen
    // graphics port.
    DrawPicture( hPicGraphics, &rectGraphics );
  }

  // We're finished with the picture, so unlock its handle and free it from
  // memory.
  HUnlock( (Handle) hPicGraphics );
  ReleaseResource( (Handle) hPicGraphics );

  // If we changes the current graphics world, set it back to what it was
  // before this routine was called.
  if ( bUsingColorGraphics )
    SetGWorld( pCGrafCurrent, hGDCurrent );

  // Restore the graphics port to what it was before this routine was called.
  SetPort( pGrafCurrent );
}
```

Drawing Text

Describing everything about how the Macintosh supports the drawing of text could fill an entire book. This means that you're going to get a crash course that leaves out many details. However, everything that you need to know about drawing text for your game will be described here.

You can specify many characteristics of text your game draws to the screen. This includes the font (Helvetica, Times, Courier, etc.), face (often referred to as style and includes underline, bold, and italics), size, and the text transfer mode. Also don't forget that the color of the text drawn is governed by the graphics port's foreground color.

When specifying which font to use, you will need to use a font number. The font number will get translated into the actual font used. The toolbox has defined font numbers for some of the most popular and standard fonts supported on Macintosh systems. Keep in mind that not all fonts may be installed on the system your game is running on. For this reason, you should try to stick to common fonts such as Helvetica, Times, Courier, Geneva, and New York. If you choose a font that's not installed on the system, the toolbox will substitute a different font for you. Also note that these are not all the fonts that you could possibly use. There is a way to determine all the fonts installed on a particular system, but that won't be discussed here.

```
enum {
     systemFont = 0,     // The system font, usually Chicago
     applFont = 1,       // The application font, usually Geneva.
     newYork = 2,
     geneva = 3,
     monaco = 4,
     venice = 5,
     london = 6,
     athens = 7,
     sanFran = 8,
     toronto = 9,
     cairo = 11,
     losAngeles = 12,
     times = 20,
     helvetica = 21,
     courier = 22,
     symbol = 23,
     mobile = 24,
};
```

When specifying the text face, you can specify one or more of the following styles as defined by the toolbox. To specify more than one style, use the logical or operator. For example, to specify bold and underline, use the expression (`bold | underline`) in the toolbox call used to change the text face. We'll see that call shortly.

```
enum {
     normal = 0,          // Normal text
     bold = 1,            // Bold
     italic = 2,          // Italics
     underline = 4,       // Underlined
     outline = 8,         // Outline
     shadow = 0x10,       // Shadow
     condense = 0x20,     // Condensed
     extend = 0x40,       // E x t e n d e d
};
```

You specify the font size in points. As anyone who has worked with a word processor knows, the higher the point size, the larger the font. A point size of 12 is typical for many applications.

Last, the text transfer mode is specified using any of the `src` transfer modes described previously. The following toolbox calls can be used to change the characteristics of the text you draw to a graphics port.

```
// Sets the text font.
void TextFont( short sFontNumber );

// Sets the text face.  Use one or more of the text styles
// above defined.
void TextFace( short sFace );

// Sets the text size.
void TextSize( short sPointSize );

// Sets the text's transfer mode.  Use an src value.  You
// will normally use srcCopy.
void TextMode( short sPointSize );
```

Okay, so now you can set all kinds of characteristics for the text you want to draw. So how do you actually draw the text to a graphics port? The first thing you need to do is position the pen to where you want the text to be drawn. The text will be drawn up and to the right of the pen. In other words, consider the pen as being located where the underline for underlined text would go. As you draw text, the pen automatically moves

to a point just after the last character of text drawn. This is done so that you can draw more text just after any previous text drawn without needing to reposition the pen. This certainly comes in handy. The following toolbox calls draw text to the current graphics port.

```
// Draw a single character at the pen's location.
void DrawChar( char ch );

// Draw a string at the pen's location.
void DrawString( Str255 str255 );

// Draw text at the pen's location.  You specify a pointer
// to the text.  sFirstChar is the offset within the text
// that's the first character to be drawn.  sCharCount
// specifies the number of characters to be drawn.
void DrawText(    Ptr         pText,
                  short       sFirstChar,
                  short       sCharCount );

// Draws text within a rectangle on the screen.  The words in
// the text are automatically wrapped at the right edge of
// the rectangle, so that the next word starts on a new line
// if it doesn't fit on the current line.  You need to
// specify a pointer to the text, its length, the rectangle
// to contain the text, and the text's justification.  The
// justifications will be described in chapter 8 on TextEdit.
// Use teJustLeft for left justified, teJustCenter for center
// justified, or teRightJust for right justified.
void TextBox(     Ptr         pText,
                  long        lTextLength,
                  Rect        *pRect,
                  short       sJustification );
```

There may be times when you need to know the width, in pixels, of the text you're about to draw. The following toolbox calls determine how many pixels wide the text specified would be if drawn to the graphics port. They consider all the text's characteristics such as font, face, and size.

```
// Determines the width of a single character.
void CharWidth( char ch );

// Determines the width of a string.
void StringWidth( Str255 str255 );
```

```
// Determines the width of the specified text.  You must
// specify a pointer to the text.  sFirstChar is the offset
// within the text that's the first character to be
// considered.  sCharCount specifies the number of characters
// to be considered.
void TextWidth(  Ptr          pText,
                 short        sFirstChar,
                 short        sCharCount );
```

The following code snippet from **Trek Window.c** draws a string that is right justified relative to a given position. In order to right-justify the text, the pen must be moved from the desired right edge by the width of the string to be drawn (since all drawing goes from left to right). Just for fun, I'm throwing in a text face change here. It is not used in the actual Desert Trek code.

```
#define STATUS_TEXT_RIGHT_EDGE 232

static StringHandle  hStringHunger;
static Rect          rectStatus[];

static void DrawFixedStrings( void )
{
  // Set the text font to Geneva.
  TextFont( geneva );

  // Set the text size to 10.
  TextSize( 10 );

  // Set the text face to be bold and underlined.
  TextFace( bold | underline );

  // Move the pen to where we want the text to go.  Since we want the text
  // to be right justified with STATUS_TEXT_RIGHT_EDGE, we need to move the
  // pen left from that edge by the length of the string we want to draw.
  // The bottom of the text is specified as rectStatus[Hunger].bottom.
  MoveTo( STATUS_TEXT_RIGHT_EDGE - StringWidth( *hStringHunger ),
          rectStatus[Hunger].bottom );

  // Draw the text.
  DrawString( *hStringHunger );
}
```

Bitmap Operations

Most games frequently need to transfer images from one bitmap to another. Whenever the screen needs to be updated, the image typically comes from an offscreen bitmap. To build that offscreen image, the graphics used often come from other offscreen bitmaps. For example, when Desert Trek needs to update the main window, the image for the main window is first built in an offscreen bitmap and then transferred to the screen in one shot. That offscreen image is really a copy of what's on the screen. To build that offscreen image, graphic elements are taken from other offscreen bitmaps. Desert Trek's desert views come from one of those bitmaps, and the 3D buttons and other graphical objects come from another offscreen bitmap. The offscreen bitmaps that hold the graphic elements used to build the copy of the game screen are set up when the program starts and remain unchanged throughout the game (unless the user changes the number of colors using the Monitors Control Panel). The offscreen bitmap that holds a copy of the main game screen gets updated each turn of the game. Figure 7.9 shows how Desert Trek's main screen gets built.

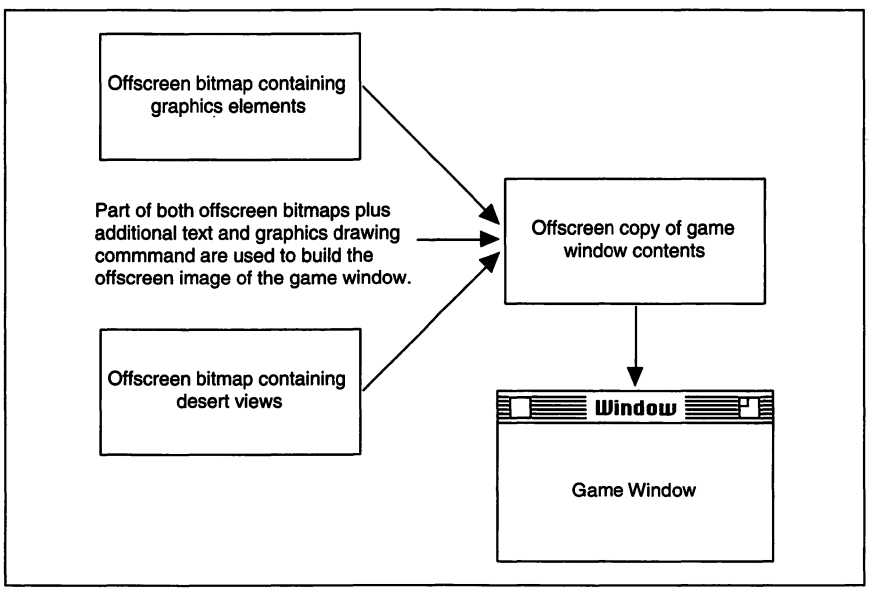


Figure 7.9 How Desert Trek's main screen gets built.

CopyBits

The Macintosh toolbox provides two very powerful functions to transfer images from one bitmap to another. Used correctly, these bitmap transfer operations can quickly and efficiently build your game's screens. The first bitmap transfer operation provided is `CopyBits()`.

```
// Copies the specified part of one bitmap into the
// specified part of another bitmap.
void CopyBits(   BitMap         *pbitmapSource,
                 BitMap         *pbitmapDestination,
                 Rect           *prectSource,
                 Rect           *prectDestination,
                 short          sTransferMode,
                 RgnHandle      hrgnClip );
```

`CopyBits()` copies the specified part of one bitmap into the specified part of another bitmap. You need to specify the source and destination bitmaps, the bounding rectangle of the part of the source bitmap you want to copy, the bounding rectangle of where in the destination bitmap you want the image copied to, the transfer mode of the operation (how bits from the source bitmap get combined with the bits of the destination bitmap), and optionally, a clipping region. The source and destination rectangles do not need to be the same size. `CopyBits()` will scale the source image to fit the destination rectangle. The transfer mode must be one of the `src` transfer modes described earlier. Most of the time you will specify `srcCopy` because you will want to replace the pixels in the destination bitmap with those from the source bitmap. You can simply specify `nil` for `hrgnClip`, unless you want to clip the operation to a particular region. Also note that `CopyBits()` can take a `PixMapHandle` as well as a `BitMap` for color pixel map transfers. In this case, if the color depth of the two pixel maps differ, `CopyBits()` supplies a conversion. Keep in mind that these conversions come at the cost of speed. More on that in a moment.

Example

The following code from **Trek Window.c** updates the main Desert Trek game window. Basically, all it does is copy the offscreen image of the

game window into the game window itself. Notice that this code works whether the offscreen graphics are contained in a black-and-white bitmap or a color pixel map. This is possible because `CopyBits()` takes either bitmap type. The variable pointing to the offscreen bitmap gets set to either the color pixel map of the graphics world or the black-and-white bitmap of the graphics port when created. In either case, this update routine doesn't need to distinguish between the two.

```
void UpdateTrekWindow( void )
{
  GrafPtr  pGrafCurrent;

  // Save the current graphics port.
  GetPort( &pGrafCurrent );

  // Set the current graphics port to the Desert Trek game window.
  SetPort( pWindowTrekWindow );

  // Begin update.
  BeginUpdate( pWindowTrekWindow );

  // Copy the offscreen bitmap containing the contents of the main game
  // window directly to the main game window.
  CopyBits( &bitmapTrekWindow, &pWindowTrekWindow->portBits,
            &rectTrekWindow, &rectTrekWindow, srcCopy, nil);

  // Draw the main game window's controls (basically, the journal's
  // scrollbar).
  DrawControls( pWindowTrekWindow );

  // End update.
  EndUpdate( pWindowTrekWindow );

  // Restore the graphics port to what it was before this routine was
  // called.
  SetPort( pGrafCurrent );
}
```

CopyMask

`CopyBits()` is a very useful way to copy rectangular images from one bitmap to another, but what if you need to copy a nonrectangular image.

For example, Desert Trek draws a palm tree growing out of a small pond when there's an oasis in the distance. The image of the palm tree needs to overlay the view of the desert. If `CopyBits()` was used, parts of the palm tree image contained within the source bounding rectangle containing information we don't want transferred to the destination bitmap will get transferred nonetheless. In other words, there is some "blank space" contained within the bounding rectangle that doesn't have anything to do with the palm tree, and that blank space will be copied to the destination bitmap. Obviously, this is a problem. What we need is a way to specify what parts of the source image contained within the bounding rectangle we want transferred to the destination bitmap. The Macintosh toolbox provides such a mechanism, called an *image mask*. An image mask is a black-and-white image where the black pixels specify what pixels in the source image you want copied to the destination bitmap. The following toolbox call allows you to copy an image from one bitmap to another and specify a mask determining which pixels from the source bitmap get copied to the destination bitmap. Figure 7.10 shows the effects of using `CopyBits()` versus `CopyMask()`.

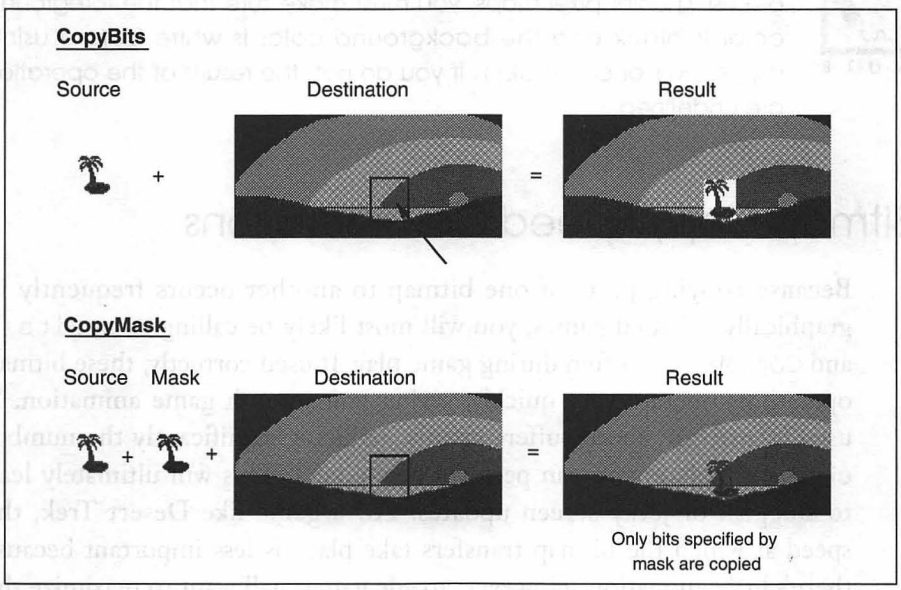


Figure 7.10 CopyBits() versus CopyMask().

```
// Copies the specified part of one bitmap into the
// specified part of another bitmap using a mask bitmap.
// The transfer mode is always scrCopy for those bits
// specified in the mask (bits not specified in the mask
// are not copied to the destination bitmap).  All three
// rectangles must be of the same size, no scaling is
// performed.
void CopyMask(   BitMap      *pbitmapSource,
                 BitMap      *pbitmapMask,
                 BitMap      *pbitmapDestination,
                 Rect        *prectSource,
                 Rect        *prectMask,
                 Rect        *prectDestination );
```

`CopyMask()` requires that all rectangles specified be the same size, which means that no scaling occurs. As with `CopyBits()`, the source and destination can be color pixel maps, but the mask bitmap must be black and white. You do not specify a clip region or a transfer mode with `CopyMask()`; the transfer mode is always `scrCopy`.

One final note about using these bitmap transfer operations. If you are using color pixel maps, you must make sure that the foreground color is black and the background color is white before using `CopyBits()` or `CopyMask()`. If you do not, the results of the operation are undefined.

Bitmap Copy Speed Considerations

Because copying parts of one bitmap to another occurs frequently in graphically oriented games, you will most likely be calling `CopyBits()` and `CopyMask()` often during game play. If used correctly, these bitmap operations operate very quickly, giving you smooth game animation. If used incorrectly, speed suffers greatly, reducing significantly the number of screen updates you can perform per second. This will ultimately lead to sluggish or jerky screen updates. For a game like Desert Trek, the speed at which the bitmap transfers take place is less important because there's little animation. However, arcade games will want to maximize the

speed at which these bitmap transfer operations take place. By adhering to the following rules, you can maximize the speed at which bitmap transfer operations take place.

1. Always, always, always make sure that the source and destination rectangles are exactly the same size. Scaling can take forever (in computer time).
2. Make certain that the source and destination pixel maps have the same color depth. Again, conversion between two color depths can take a very long time.
3. It is much better to perform a small number of large bitmap transfers than it is to perform a large number of small bitmap transfers. This is due to the fact that the toolbox incurs a lot of overhead just setting up a bitmap transfer.
4. Whenever possible, use the same rectangle to specify the exact same coordinates in both the source and destination bitmaps. This is easily accomplished for bitmaps that are a copy of a window on the screen, because both the window and offscreen bitmaps can have the same origin and size.

Drawing Directly to the Screen

Some games bypass `CopyBits()` and draw directly to the screen for speed. Don't do this. Don't even think about doing this. By doing so, you guarantee that your game will be incompatible with some video cards and monitors out there. This assumes that you can even get it to work reliably on standard Macintosh hardware. In addition, if your direct-to-screen code is written in 680x0 assembly, as was the common method to implement direct-to-screen drawing in the past, your game will actually run slower on Power Macintoshes than if you had used `CopyBits()` in the first place. This is because the native 680x0 code would have to be interpreted, whereas the `CopyBits()` code is native to the platform it's running on, be it a 680x0-based Macintosh or a Power Macintosh.

Desert Trek's View Transition Special Effect Example

Desert Trek provides a special graphical effect when the view in the game window changes. Basically, the old view dissolves into the new view. This is accomplished by using multiple patterns and `CopyMask()`. The new view is copied into the Desert Trek game window in parts. The parts copied are defined by 16 patterns, as seen in Figure 7.11. Each pattern gets drawn one at a time into a black-and-white offscreen graphics port that's the same size as the view portion of the Desert Trek game window. That offscreen graphics port is then used as a mask for copying the new view image into the game window. This means that only selected pixels from the new view get copied onto the screen. This process is repeated for a total of 16 times where each time, more and more pixels get copied to the screen. Eventually, all the pixels are copied and the special effect is complete. The following code can be found in **Trek Window.c**.

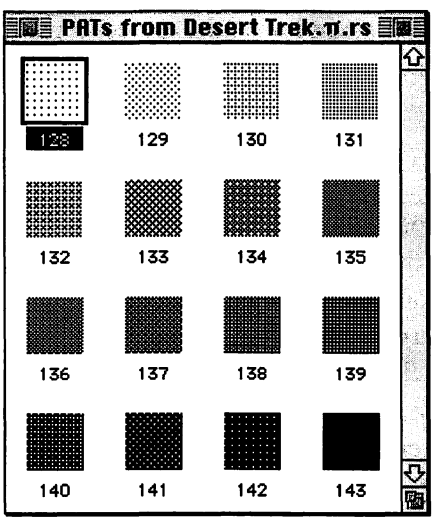
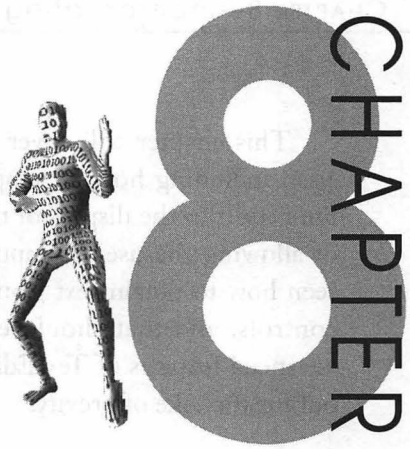


Figure 7.11 The pattern used for the view transition.

```
void DrawViewTransition( void )
{
  Boolean   bUsingColorGraphics;
  short     sLoop;
  GrafPtr   pGrafCurrent;

  // Save the current graphics port so that it can be restored later.
  GetPort( &pGrafCurrent );

  // Determine if we are using color graphics.
  bUsingColorGraphics = UsingColorGraphics( nil );

  // Make sure that the Desert Trek game window is updated with the current
  // game status, including the view of the player's current location.
  UpdateTrekWindow();

  // Draw the new view offscreen.  It is this new view that will get "phased"
  // into the game window.
  DrawViewOffscreen( nil, nil, bUsingColorGraphics, nil );

  // If we are using color graphics, lock down the pixel map of the Desert
  // Trek game window.  Remember, you must lock a pixel map before drawing
  // to it.
  if ( bUsingColorGraphics )
    LockPixels( pGWorldTrekWindow->portPixMap );

  // Set the current graphics port to the offscreen graphics port that will
  // be filled with the patterns used to phase the new view onscreen.  This
  // graphics port is the same size as the view portion of the game window.
  SetPort( pGrafViewPatterns );

  // Loop through each pattern.  The first pattern has very few bits set, and
  // the last pattern is solid black.
  for( sLoop = 0; sLoop < NUMBER_VIEW_PATTERNS; sLoop++ )
  {
    // Fill the mask bitmap with the specified pattern.
    FillRect( &rectView, **hPatViewPatterns[sLoop] );

    // Copy only those pixels from the new view specified by the mask pattern
    // above into the view part of the Desert Trek game window.  The first
    // time through this loop, only a few pixels from the new view will get
```

```
    // copied to the view in the game window.  Each time though this loop,
    // more and more of the new view will get copied into the game window
    // until the last copy, which will copy the entire new view into the
    // game window.
    CopyMask( &bitmapTrekWindow, &bitmapViewPatterns,
              &pWindowTrekWindow->portBits,  &rectView, &rectView,
              &rectView );

    // Delay for 2/60th of a second.  Otherwise, if Desert Trek is running on
    // a really fast Macintosh, the special effect would go by so quickly that
    // the user wouldn't get a chance to see it.
    NiceDelay( 2 );
  }

  // If we locked the Desert Trek game window's pixel map, unlock now since
  // we are done with it.
  if ( bUsingColorGraphics )
    UnlockPixels( pGWorldTrekWindow->portPixMap );

  // Restore the current graphics port back to what it was before this
  // routine was called.
  SetPort( pGrafCurrent );
}
```



INCORPORATING TEXT

There will be many times when you'll want to display considerable quantities of text in your game. For example, if your game supports on-line help, you will need to give the user the ability to view and scroll through the help text. Desert Trek also maintains a textual journal that contains a description of the player's journey across the desert. It should be fairly clear that using the somewhat limited text drawing routines described in Chapter 7 on QuickDraw won't provide enough functionality when it comes to displaying, formatting, and scrolling the amount of text involved for these functions. The Macintosh toolbox provides the TextEdit manager to deal with such quantities of text. As the name implies, TextEdit gives Macintosh applications the ability to allow for the entering of text as well as the display of text.

This chapter will cover many aspects of the TextEdit toolbox manager, including how to support styled text. However, this chapter will limit itself to the display of textual information and not cover the process of allowing the user to input text into text edit fields. We have already seen how to obtain text from the user in Chapter 6 on dialog boxes and controls, and that should easily suffice for most games. Also, certain advanced features of TextEdit, rarely used by game programs, will be left out for the sake of brevity.

Text Edit Records

Just as windows and controls have records defining their properties, text edit fields have records that define their properties. Unlike the case with windows and controls, however, the toolbox does not provide a method to access all of those fields with toolbox calls. In the case of text edit fields, then, you will need to know the names of some of the fields contained within the text edit record.

You should also note that there are really two types of text edit records. They contain the same number of fields, and the field names are all the same, but the contents of some of the fields differ. These two types of text edit records are commonly referred to as the "old style" text edit record, and the "new style" text edit record. The fundamental difference between the two text edit records is that the new style text edit record supports different text styles within the text edit record. The old style text edit record only supported one text style which applied to all the text in the text edit record. The new text edit record allows you to define multiple styles for different sections of text. In order to support the new style text edit record while at the same time supporting applications that used the old style text edit record, the toolbox interprets some of the fields within the text edit record differently. You need not be concerned over these differences.

The following fields within the text edit record may interest your game. The only way to obtain the information they contain is to directly

access the field of the text edit record itself. Do not change their values, though. Doing so could cause major problems, and simply begs for trouble.

```
struct TERec {
    Rect    destRect;              // Destination rectangle.
    Rect    viewRect;              // View rectangle.
    .
    .
    .
    short   teLength;              // Number of characters.
    .
    .
    .
    short   nLines;                // Number of lines.
    short   lineStarts[16001];     // First character of each line.
};

typedef struct TERec TERec;
typedef TERec *TEPtr, **TEHandle;
```

destRect: The destination rectangle specifies the entire rectangle bounding all the text within a text edit record. Note that this rectangle can easily be much larger than what can be displayed on the screen. As text is added to or deleted from the text edit record, the destination rectangle's height grows or shrinks accordingly. The destination rectangle's width determines how words are broken to form lines. Text edit records automatically break lines of text for you to fit within the width of this rectangle. In other words, if a word extends beyond the right edge of the destination rectangle, the word will instead be placed at the left edge of the next line.

The top of the destination rectangle is not constant. It reflects the portion of the text edit record currently being viewed in the graphics port. As you scroll the text of a text edit record up or down, the top and bottom of the destination rectangle changes accordingly. This means that you will use the top of the destination rectangle to determine how you need to scroll the text to match the value of the scroll bar associated with that text edit record. We'll see an example of this later in this chapter.

viewRect: This field specifies where the text edit record is located within the graphics port. You need to supply this rectangle as a parameter when creating a text edit record. All text associated with the text edit record is displayed within this rectangle.

teLength: This field tells you how many characters are contained within the text edit record. Note that a text edit record can contain a maximum 32,000 characters. You will need to check this field whenever you add text to the text edit record to make sure you do not exceed the 32,000-character limit.

nLines: This field tells you how many lines are contained within the text edit record. You will need to use this field to determine what maximum and minimum control values to set for the scroll bar associated with the text edit record. In other words, the number of lines in the text edit record directly determines how much scrolling the user needs to do in order to see all the text.

lineStarts[]: This field gives you the offset of the first character of each line. The offset is expressed in number of characters from the beginning of the text contained within the text edit record. As we'll see later, this field is useful for determining what line a particular character is on, or for deleting a number of lines from the text edit record.

Creating and Destroying Text Edit Records

You do not need to allocate storage for a text edit record when creating one because the toolbox takes care of all the memory management for you. This also includes the management of the memory taken up by the text and text styles contained within the text edit record.

Certain characteristics of the text contained within a text edit field are determined at the time of creation. In other words, the text's font, style, size, and transfer mode get copied from the corresponding characteristics of the current graphics port when the text edit record is created.

Make sure to set these text characteristics before creating the text edit record. This is especially true if you plan to use an old style text edit record because you can't change the text's characteristics after the text edit record is created.

The following two toolbox calls create either an old style text edit record or a new style text edit record. The text edit record will be created inside the current graphics port, be it an offscreen graphics port, or an onscreen window. These toolbox routines return a handle to the text edit record created. You will need to use this handle when referencing that text edit record, so keep it handy.

```
// Creates an old style text edit record, one which will not
// support multiple text styles.
TEHandle TENew(      Rect        *pRectDestination,
                     Rect        *pRectView );

// Creates a new style text edit record, one which will
// support multiple text styles.
TEHandle TEStylNew(  Rect        *pRectDestination,
                     Rect        *pRectView );
```

The two rectangle parameters are specified in the local coordinates of the graphics port in which the text edit record is created. The first rectangle, `pRectDestination`, is really used to specify the width of a text edit record. The height of this rectangle will grow or shrink automatically as text is added to or deleted from the text edit record. The second rectangle, `pRectView`, is the *viewing* rectangle. The viewing rectangle is where text will be displayed in the graphics port containing the text edit record. Most of the time you will want the destination rectangle to be the same as the view rectangle when you first create a text edit record. Unless you plan to support the horizontal scrolling of text, make sure that both rectangles have the same width.

Once you are finished with a text edit record, you need to dispose of it. This causes it to be removed from its graphics port and frees all its memory resources. The following toolbox call disposes of both old and new style text edit records.

```
// Disposes of a text edit record, removing it from the
// graphics port and freeing up all its memory.
void TEDispose( TEHandle hTE );
```

Updating Text Edit Records

When a window receives an update event, you need to update all text edit records contained within that window. In addition, anytime you change the appearance of a text edit record, you need to update it so that those changes get reflected in the graphics port or window containing that text edit record. The following toolbox call updates a text edit record. Note that `TEUpdate()` is automatically called for you whenever you add, delete, or scroll the text within a text edit record.

```
// Redraws the portion of a text edit record specified by
// pRectUpdate.
void TEUpdate(   Rect        *pRectUpdate,
                 TEHandle    hTE );
```

Text Justification

You can justify text within a text edit record to be either left justified, centered, or right justified. The justification affects all text within the text edit record, meaning that you cannot justify sections of text within the same text edit record differently. The following justification constants are defined by the Macintosh toolbox.

```
enum {
      teJustLeft = 0,      // Left justified text.
      teJustCenter = 1,    // Centered text.
      teJustRight = -1     // Right justified text.
}
```

To set the text justification, use the following toolbox call.

```
// Sets the text justification for the specified text edit
// record.
void TESetJust( short       sJustification,
                TEHandle    hTE );
```

Line Height

There will be times when you need to determine the height of one or more lines in the text edit record. For example, if you need to determine how many lines fit on the screen at one time, something which you'll need to do to set the scroll bar's control values correctly, you have to compute the average height of a line in the text edit record. The average height of a line is the total height of all lines divided by the number of lines in a text edit record. The following toolbox call returns the height, in pixels, of the range of lines specified. If you want the height of a single line, use the same value for the start and end lines. *Note that you specify the end line first.*

```
// Returns the height, in pixels, of the lines specified.
long TEGetHeight( long        lEndLine,
                  long        lStartLine,
                  TEHandle    hTE );
```

Character Coordinates

As mentioned earlier, you can imagine a single large rectangle that bounds all of the text contained within a text edit record. At times, it might be useful to determine which character lies at a particular point within that bounding rectangle, or, to determine the point within that bounding rectangle at which a particular character lies. Desert Trek uses this information to support the display of pictures in a text edit record. The position of the picture is specified relative to a particular character in the 'TEXT' resource. The location of a picture within the text edit record's bounding rectangle can be computed using the coordinates of

the character next to that picture. We'll see more about the support of pictures in text edit records toward the end of this chapter.

The following two toolbox calls can be used to convert between a character's offset within the text and its point coordinate within the text edit record's bounding rectangle.

```
// Returns the character offset within the text edit record's
// text that corresponds to the point specified.
short TEGetOffset(   Point        pt,
                     TEHandle     hTE );

// Returns the point location corresponding to the character
// offset within the text edit record's text.
Point TEGetPoint(    short        sCharacterOffset,
                     TEHandle     hTE );
```

Selecting Text

Before we get into how to add and delete text from a text edit record, it is important to know how to select text within a text edit record. Why? Well, when adding text to a text edit record, the text selection determines where that text gets added. Similarly, to delete text from a text edit record, you must first select the text you want deleted.

To select text in a text edit record, use the following toolbox call. Note that the selected text gets highlighted or inverted (e.g., white text on a black background, as opposed to the usual black text on a white background).

```
// Sets the selected text within the specified text edit
// record.  The selection is specified by character position
// within the text edit record.  If the start and end
// selections are the same, you are really setting the
// insertion point's location.
void TESetSelect(    long         lSelectionStart,
                     long         lSelectionEnd
                     TEHandle     hTE );
```

The start and end selection characters are specified as an offset from the first character in the text edit record. If you try to set the text selection

beyond the last character of the text edit record, the selection will stop at the last character. If the start character specified occurs after the end character specified, the two parameters are automatically reversed. If you want to deselect all the text, or set the insertion point for adding text, specify the same value for the start and end selection characters. This will cause the insertion point to be positioned just after the character specified. If you want to position the insertion point just before the first character, specify 0 for both the start and end positions.

Adding Text

To insert text into a text edit record, use one of the following two toolbox calls. You need to specify a pointer to the text that is to be added, the length of the text that is to be added, and the text edit record to which you are adding the text. For new style text edit records, you can also specify an array of style tables to be used to set the styles of the text added. Most of the time, this array will be loaded from a resource of type 'styl'.

```
// Inserts text into the specified text edit record.  The
// text gets inserted at the insertion point or just before
// the first character selected in the text edit record.
void TEInsert(      Ptr           pText,
                    long          lTextLength,
                    TEHandle      hTE );

// Inserts text into the specified text edit record.  The
// text gets inserted at the insertion point or just before
// the first character selected in the text edit record.  The
// styles pointed to by hST will be used to set the text
// styles of the text inserted, and is typically loaded from
// a 'styl' resource.
void TEStylInsert( Ptr           pText,
                   long          lTextLength,
                   stScrpHandle  hST,
                   TEHandle      hTE );
```

The text to be inserted gets added at the insertion point. If there's a block of text selected, the insertion point is considered to be located just before

the first character of the selection. The selected text remains unaffected after inserting text. In other words, it remains selected. If you want the inserted text to replace the selected text, you'll need to delete the selected text first. We'll see how to do that in the next section.

The following example adds styled text to a new text edit record. The text is loaded from a resource of type 'TEXT', and the styles table associated with that text is loaded from a resource of type styl. When you create a 'TEXT' resource using ResEdit, a styl resource with the same ID is created to contain the styles you select while editing the text. The following code excerpt comes from **Information Window.c**. Only the code associated with the text edit record is shown here.

```
void  ConstructInfoWindow( short    sWindowID,
                           short    sTextResourceID,
                           short    sTopicsMenuID,
                           Boolean  bUseColor )
{
  PINFO_WINDOW  pInfoWindow;
  Handle        hText;
  StScrpHandle  hStScrpText;
  Handle        hItem;
  short         sMaxScrollValue;
  Rect          rect;
  short         sItemType;

  // I created a user item in the dialog to specify the location of the text
  // edit record within that dialog.  Here I get it's rectangle so I know what
  // to specify as the text edit record's view rectangle.
  GetDItem( pInfoWindow->pDialog, INFO_WINDOW_TE_ID, &sItemType, &hItem,
            &(pInfoWindow->rectText) );

  // Create a new text edit record that supports multiple text styles.
  pInfoWindow->hTE = TEStylNew( &(pInfoWindow->rectText),
                                &(pInfoWindow->rectText) );

  // Load the text for the text edit record from the resource fork.
  hText = GetResource( 'TEXT', sTextResourceID );

  // Load the associated text styles array from the resource fork.
  hStScrpText = (StScrpHandle) GetResource( 'styl', sTextResourceID );
```

```
    // Lock the text in memory.
    HLock( hText );

    // Insert the text and its associated text styles into the text edit record.
    TEStylInsert( *hText, SizeResource( hText ), hStScrpText,
                  pInfoWindow->hTE );

    // Done with the text and style array.  Unlock and free them from memory.
    HUnlock( hText );
    ReleaseResource( hText );
    ReleaseResource( (Handle) hStScrpText );

    // Compute how many lines fit on the screen at one time.  This is determined
    // using the height of the display area and dividing it by the average
    // height of a line in the text edit record.  The average height of a line
    // in the text edit record is computed by dividing the entire height of all
    // the lines in the text edit record by the number of lines in the text edit
    // record.
    sHeight = TEGetHeight( (*pInfoWindow->hTE)->nLines, 0, pInfoWindow->hTE );

    pInfoWindow->sLinesPerPage = ( pInfoWindow->rectText.bottom -
                                   pInfoWindow->rectText.top ) /
                                 ( sHeight / (*pInfoWindow->hTE)->nLines );

    // Compute the scrollbar's maximum value.  This is the total number of lines
    // in the text edit record minus the number of lines that can fit on the
    // screen at one time.
    sMaxScrollValue = (*pInfoWindow->hTE)->nLines - pInfoWindow->sLinesPerPage;

    // If all the lines fit on the screen, we can disable the scrollbar...
    if ( sMaxScrollValue <= 0 )
      HiliteControl( pInfoWindow->hControlScrollbar, 255 );

    // otherwise set the scrollbar's maximum value.
    else
      SetCtlMax( pInfoWindow->hControlScrollbar, sMaxScrollValue );
}
```

Deleting Text

There may be times when you need to delete text from a text edit record. The following toolbox call deletes the selected text from the specified

text edit record. The insertion point will be left where the deleted text used to be, and of course, there will no longer be any selected text.

```
// Deletes the selected text from the specified text edit
// record.
void TEDelete( TEHandle hTE );
```


TIP

If you are constantly adding text to the end of a text edit record, such as Desert Trek does for keeping the game's journal, you need to be careful not to exceed the 32,000 character limit. Otherwise, some of the text you're adding won't really get added. A great way to make sure the text you're adding really gets added is to check to make sure there's enough room for that text. If not, delete enough text from the beginning of the text edit record to make room for the new text. The following routine from Desert Trek adds text to the game's journal. Notice that it deletes lines of text from the beginning of the journal if there isn't enough room to hold the next text. This routine can be found in **Journal.c**.

```
void AddJournalText( char    *szText,
                     short    sLength,
                     Boolean  bUpdateWindowNow )
{
  short    sMaximumScrollValue = 0;

  // While there isn't enough room in the text edit record to hold the new
  // new text, delete the first line of the text edit record.
  while ( (*teJournal)->teLength + sLength >= 32000 )
  {
    // First select the text we want to delete.  In this case, it's the entire
    // first line of text in the journal.
    TESetSelect( 0, (*teJournal)->lineStarts[1], teJournal );

    // Delete the selected text.
    TEDelete( teJournal );
  }

  // We always want the new text to be added at the end of the journal.  So,
  // we must set the insertion point to the end of the journal.  Do so by
  // setting the selection start and end positions to be the last character
  // of the journal.
  TESetSelect( (*teJournal)->teLength, (*teJournal)->teLength, teJournal );
```

```
    // Insert the new text into the journal.
    TEInsert( szText, sLength, teJournal );

    // Recompute the maximum scrollbar value based on the new number of lines
    // in the journal (and the number of lines displayed on the screen).
    sMaximumScrollValue = (*teJournal)->nLines - sLinesPerPage;

    // Make sure that the scrollbar's maximum value at least 0.
    if ( sMaximumScrollValue <= 0 )
      sMaximumScrollValue = 0;

    // Set the scrollbar's highlight state.  Remember, if there's nothing to
    // scroll, we need to disable the scrollbar.  Otherwise, it needs to be
    // enabled.
    HiliteControl( hControlJournalScrollbar, sMaximumScrollValue ? 0 : 255 );

    // Set the new maximum value for the scrollbar, as computed above.
    SetCtlMax( hControlJournalScrollbar, sMaximumScrollValue );

    // Whenever text is added to the journal, we always want to scroll the text
    // viewed by the user to the bottom of the text edit record.  This will
    // force the new text added to be displayed to the user, regardless of what
    // journal text they were looking at before the new text was added.
    SetCtlValue( hControlJournalScrollbar, sMaximumScrollValue );

    // Update the Desert Trek main window if specified by the calling routine.
    if ( bUpdateWindowNow )
      SynchJournalTextWithScrollbar();
}
```

Setting Text Style

As stated above, new style text edit records support multiple text styles for different sections of text. What styles can you set for the text? The toolbox defines the following text style structure that changes the text's style.

```
struct TextStyle {
    short       tsFont;       // Font number.
    Style       tsFace;       // Face style (bold, italics, etc.)
    char        filler;
    short       tsSize;       // Text point size.
```

```
    RGBColor    tsColor;        // Text RGB Color.
};

typedef struct TextStyle TextStyle;
typedef TextStyle *TextStylePtr, **TextStyleHandle;
```

How do you set the style for a section of text within a text edit record? First, you need to create a text style structure and populate it with the desired style values. Then you need to select the text that you want to affect. After doing so, use the following toolbox call to set the selected text's style.

```
 // Modes used when calling TESetStyle().
enum {
    doFont = 1,     // Change font.
    doFace = 2,     // Change face (bold, italics, etc.)
    doSize = 4,     // Change size.
    doColor = 8,    // Change color.
    doAll = 15      // Change them all!
}

// Sets the text style of the selected text within the
// specified text edit record.  The sMode parameter specifies
// which attributes to set.  You can add the modes together
// if you want to modify multiple text attributes.  Set
// bRedraw to true to have the text immediately redrawn.
void TESetStyle( short          sMode,
                 TextStylePtr   ptextStyle,
                 Boolean        bRedraw,
                 TEHandle       hTE );
```

If you need to determine which styles apply to a given character within a text edit record, you can use the following toolbox call to obtain it.

```
// Obtains style information for the character at the given
// offset within the text edit record's text.  Also returns
// the height of the line containing that character as well
// as the font ascent for that character.
void TEGetStyle(  short           sCharacterOffset,
                  TextStylePtr    ptextStyle,
                  short           sLineHeight,
                  short           sFontAscent,
                  TEHandle        hTE );
```

Scrolling Text

How do you make sure that the correct portion of a text edit record is being displayed on the screen? The main point of using a text edit record is to allow the user to scroll through a large amount of text. It is up to you to make sure that the text corresponding to the scroll bar's value is being displayed on the screen. The toolbox provides the following routine to scroll text within a text edit record.

```
// Scrolls a text edit record horizontally and/or vertically
// by the amounts specified.
void TEScroll(  short           sHAmount,
                short           sVAmount,
                TEHandle        hTE );
```

In essence, this routine modifies the destination rectangle, offsetting it by the horizontal and vertical values you specify. Positive values for the offsets cause text to scroll down or to the right, negative values cause the text to scroll up or to the left. Usually, you will only be concerned with scrolling text up or down, meaning that you can specify 0 for the horizontal scroll amount.

The toolbox also provides a quick method to scroll the selected text into view. However, if you use this call, you'll need to set the scroll bar's value manually. Remember, you usually scroll the text in a text edit record to match the value of the scroll bar associated with that text edit record. If you start scrolling the text yourself, you'll need to make sure the scroll bar stays synchronized.

```
// Scrolls the selected text into view.
void TESelView( TEHandle hTE );
```

Example

The following example from Desert Trek synchronizes a text edit record's text with the current scroll bar value. Whenever the user changes the value of the scroll bar, the text associated with the scroll bar's value needs to be scrolled into view. The following routine from **Journal.c** scrolls the journal text accordingly.

```
void SynchJournalTextWithScrollbar( void )
{
  GrafPtr  pGrafCurrent;
  short    sCurrectPosition;
  short    sNewPositon;
  short    sScrollbarValue;

  // Save the current graphics port so that we can restore it later.
  GetPort( &pGrafCurrent );

  // Set the current graphics port to the journal's graphics port.
  SetPort( pGrafJournal );

  // Lock the text edit record, so it stays fixed in memory while we're using
  // it.
  HLock( (Handle) teJournal );

  // Get the current vertical position of the text being viewed on the screen.
  // It's the top of the view rectangle minus the top of the destination
  // rectangle.  Remember, it's the destination rectangle that changes when
  // we scroll a text edit record's text.
  sCurrectPosition = (*teJournal)->viewRect.top - (*teJournal)->destRect.top;

  // Get the scrollbar's current value.
  sScrollbarValue = GetCtlValue( hControlJournalScrollbar );

  // If the scrollbar's value is 0, the user is looking at the first page of
  // text.  Thus, the new position is 0, meaning the top of the text.
  if ( !sScrollbarValue )
    sNewPositon = 0;

  // Otherwise, the position we need to scroll to is determined by the
  // scrollbar's value.  Remember that the scrollbar tells us which line
  // is the first line being viewed on the screen.  Thus, we need to scroll
  // to the vertical coordinate specified by that line.  TEGetHeight() tells
  // us that value.
  else
    sNewPositon = TEGetHeight( sScrollbarValue, 0, teJournal );

  // Scroll the text to the new current position.  Since TEScroll takes an
  // offset, we need to subtract the new position from the current position.
  // Note that we don't want to scroll the text horizontally, so the first
  // offset specified is 0.
  TEScroll( 0, sCurrectPosition - sNewPositon, teJournal );
```

```
    // We're done with the text edit records, so unlock it.
    HUnlock( (Handle) teJournal );

    // Update the window to reflect the scrolled text.  The journal's text edit
    // record is really kept in an offscreen graphics port, and then copied to
    // the screen using CopyMask().
    DrawJournalInTrekWindow();

    // Restore the graphics port to what it was before this routine was called.
    SetPort( pGrafCurrent );
}
```

Accessing Text

There will be times when you need to access the text of a text edit record. For example, Desert Trek can save the journal's text into a text file. In order to do so, Desert Trek needs to access the text within the journal's text edit record so that it can be saved to a file. The following toolbox routine returns a text edit record's text.

```
typedef char Chars[32001];
typedef char *CharsPtr,**CharsHandle;

// Returns a character handle to the text contained within
// the specified text edit record.
CharsHandle TEGetText( TEHandle hTE );
```

Be very careful with the character handle returned, because it points to the actual text contained within the text edit record. Do not modify the text pointed to by the character handle, otherwise you'll be asking for trouble. Also note that if you dispose of a text edit record, any character handles you have pointing to that text edit record's text will become invalid because that text will be disposed of with the text edit record.

Searching for and Replacing Text

The toolbox provides a very powerful text search and replace routine. You can use this routine to search for a string in a text edit record and, option-

ally, replace it with another string. There's no reason why you can't use the following routine on blocks of text not contained within text edit records, but generally, it is used on text contained within text edit records.

```
// Searches text for a target string and optionally replaces
// the target string with a replacement string.  Returns a
// negative value if the target string is not found.
long Munger(   Handle    hText,
               long      lStartPosition,
               Ptr       pTargetText,
               long      lTargetLength,
               Ptr       pReplacementText,
               long      lReplacementLength );
```

`Munger()` returns an offset into the text block or a negative value upon completion. A negative value means that the target text was not found. A non-negative result means that the target text was found, and if specified, replaced. In the case where no replacement text was specified, the return value is the character position just after the found string. If replacement text was specified, the return value is the character position just after the text replacing the target string.

`Munger()` takes quite a few parameters, so lets take a closer look at them. You first need to specify a handle to the text to be searched. For a text edit record, this is the character handle returned by `TEGetText()`. Next, you need to specify the starting position of the search. If you want to start the search at the beginning of the text block, specify 0. Combined with the return value, this parameter is very useful when you are doing a global search and replace. You simply call `Munger()` repeatedly, passing it the return value of the previous call until you finally get a negative result, that is, no more occurrences of the text string can be found.

The text string you want to look for is specified by a pointer to that string. This is called the *target text* because that's the target of your search. You need to specify the length of the target text, which means that you can't assume it's a standard C null-terminated string. It is legal to specify `nil` as the target, which means that text will always get replaced starting at the `lStartPosition` character in the text array. The number of characters replaced in this case is specified in the

`lTargetLength` parameter. If you specify 0 for the target text length, the replacement text gets inserted at the specified starting position.

The replacement string is also specified by providing a pointer to that string and its length. If a replacement string is specified, the found target string will be replaced with the replacement string. The strings do not need to be the same length. It is perfectly legal to specify `nil` as the replacement string, in which case a found target string is not replaced. This is useful if you are only interested in locating a string within a text block. One final note: If you specify a replacement string and a replacement string length of 0, the target string is simply deleted (since the replacement string has a length of 0).

Drawing Pictures in Text Edit Records

The following code examples show you how you can support graphics in text edit records. Basically, you need to do two things: (1) you need to determine where the graphics go in relation to the text contained within the text edit record, and (2) you need to draw the graphics on the screen when the user has scrolled them into view.

This is really very simple to accomplish. Think about a large bounding rectangle that encompasses all the text of a text edit record. Now, envision pictures placed among the text contained in that bounding rectangle. How do you determine where the pictures go? What you really need to do is to calculate a rectangle bounding the picture. You'll know the picture's width and height from the picture itself, so all you really need is the corner of the rectangle within the text edit record that will bound that picture. That corner can be calculated by specifying the picture's location relative to a character's position in the text. For example, you can say that picture 1 is located after character 86. After setting the pictures' positions, it's easy to determine what pictures to draw based on the scroll bar's value. You're already doing this to determine what text to display based on the scroll bar's value.

Desert Trek determines what character a picture is drawn near via a special keyword specified in the 'TEXT' resource. In fact, Desert Trek

provides three special keywords that can be placed in the 'TEXT' resource: one for icons, one for pictures, and one that determines where to scroll when a topic is selected from the topic's pop-up menu. If you want to use the code from **Information Window.c** to support help and other information windows in your game, you can specify these keywords anywhere in the 'TEXT' resource.

The >*Topic*< keyword determines which line of the 'TEXT' resource gets displayed at the top of the screen when a topic is selected from the topic's pop-up menu. If the user selects the first topic from the topic's pop-up menu, the text gets scrolled to the first occurrence on the >Topic< keyword. If the user selects the second topic from the topic's pop-up menu, the text gets scrolled to the second occurrence of the >Topic< keyword. You can probably take things from here.

The >*Pict(p1,p2,p3,p4)*< keyword is used to place a picture in the text edit record at the position of the keyword. Make sure to leave enough blank lines in the text to allow space for the picture, otherwise it will overlay some of the text. The Picture keyword takes four parameters. The first parameter, p1, specifies the minimum color depth needed to support a color picture. For example, 4 means that the monitor must be set to at least **16** color mode in order to use a color picture. A value of 8 would correspond to 256 colors. The second parameter, p2, specifies the resource ID of the color picture to draw at that location in the text edit record. The third parameter, p3, specifies the resource ID of the black-and-white picture to draw at that location if the monitor isn't set to the color depth specified by parameter p1. The final parameter, p4, specifies the justification of the picture within the text edit record. A value of 1 means left-justified, a value of 2 means centered, and a value of 3 means right-justified.

The >*Icon(p1,p2)*< keyword places an icon in the text edit record at the position of the keyword. Again, you'll need a couple of black lines to leave space to draw the icon. The first parameter, p1, specifies the resource ID of the icon, which can be either a color icon of type cicn, or a black-and-white icon of type 'ICON'. If you have both types of icons defined, the proper icon will be drawn based on the monitor's pixel depth. The second parameter, p2, specifies the justification. Again, use 1 for left-justification, 2 for center-justification, and 3 for right-justification.

You can look at the code in **Information Window.c** to see how the 'TEXT' resource gets parsed. Basically, `Munger()` is used to locate all three types of keywords. The keywords are removed from the text before the text gets put into the text edit record so that the user doesn't see them. The following code from **Information Window.c** determines the location of a picture based on which character in the text edit record the picture follows. Again, think of this position as being located somewhere within the large rectangle bounding the entire text edit record.

```
static void SetPicturePosition( PINFO_WINDOW  pInfoWindow,
                                PPICTURE_INFO pPictureInfo,
                                Rect          *prectBitmap )
{
  short  sLineNumber;
  short  sLineHeight;
  short  sLeft;
  short  sTop;
  Point  ptCorner;

  // Find the line number of the line containing the character at which this
  // picture is to be located.  This is a Desert Trek routine.
  sLineNumber = FindLineNumber(pInfoWindow, pPictureInfo->sCharacterPosition);

  // Get the height of the line containing this picture.
  sLineHeight = TEGetHeight( sLineNumber, sLineNumber, pInfoWindow->hTE );

  // Get the point location within the text edit record's bounding rectangle
  // that corresponds to the character's position.  This is where the picture
  // will be located within the bounding rectangle of the text edit record.
  ptCorner = TEGetPoint( pPictureInfo->sCharacterPosition, pInfoWindow->hTE );

  // Compute the vertical coordinate of the picture.  We need to subtract the
  // height of the line from the vertical position computed above to account
  // for the fact that the point returned by TEGetPoint() is really the bottom
  // of the character's position within the text edit record.
  sTop = ptCorner.v - sLineHeight;

  // If the picture is left justified, it's horizontal position is at the left
  // of the text edit record (plus a little margin).
  if ( pPictureInfo->lJustification == 1 )
    sLeft = pInfoWindow->rectText.left + 2;
```

```
    // If the picture is centered, it's horizontal coordinate is the middle of
    // the text edit record minus half the picture's width.
    else if ( pPictureInfo->lJustification == 2 )
      sLeft = pInfoWindow->rectText.left + ( pInfoWindow->rectText.right -
              prectBitmap->right ) / 2;

    // If the picture is right justified, the left edge of the picture is the
    // right edge of the text edit record minus the width of the picture.
    else
      sLeft = pInfoWindow->rectText.right - prectBitmap->right - 2;

    // Set's the picture's location within the text edit record.
    SetRect( &pPictureInfo->rectPosition, sLeft, sTop,
             sLeft + prectBitmap->right, sTop + prectBitmap->bottom );
}
```

After you know where the pictures go, you simply need to draw them on the screen when appropriate. Again, it's as simple as knowing what part of the rectangle bounding the entire text edit record is being displayed on the screen. If any pictures are located within the part being displayed on the screen, you need to draw them. The following routine from **Information Window.c** determines if a picture is visible on the screen and if so, draws it. This routine gets called whenever the value of the scroll bar changes.

```
static void DrawPictures( PINFO_WINDOW pInfoWindow )
{
  short        sLoop;
  short        sScrollbarValue;
  short        sCurrentHeight = 0;
  Rect         rectView;
  Rect         rectSect;
  Rect         rectSectPicture;
  PPICTURE_INFO pPictureInfo;

  // Lock the handle to the picture information records so that we can
  // assign a pointer to it which can be used throughout this function.
  // This will speed access to the picture information records.
  HLock( (Handle) pInfoWindow->hPictures );
  pPictureInfo = *pInfoWindow->hPictures;

  // Get the current scrollbar's value so we can use it to calculate the part
```

```
// of the text edit record currently being displayed on the screen.
sScrollbarValue = GetCtlValue( pInfoWindow->hControlScrollbar );

// If the scrollbar's value is not 0 (meaning that the user is not looking
// at the top of the text edit record), determine the vertical coordinate of
// the line currently displayed at the top of the screen.
if ( sScrollbarValue )
  sCurrentHeight = TEGetHeight( sScrollbarValue, 0, pInfoWindow->hTE );

// Set the rectangle specifying what part of the text edit record is being
// displayed on the screen.  This rectangle is relative to the top left of
// the rectangle bounding the entire text edit record.  The left and right
// coordinates are simply the left and right edges of the text edit record
// itself.
SetRect( &rectView, pInfoWindow->rectText.left, sCurrentHeight +
        pInfoWindow->rectText.top, pInfoWindow->rectText.right,
        sCurrentHeight + pInfoWindow->rectText.bottom );

// Loop through each picture contained in this text edit record to see
// if they intersect the rectangle being viewed on the screen.
for( sLoop = 0; sLoop < pInfoWindow->sNumberPictures;
     sLoop++, pPictureInfo++ )

  // If the picture's rectangle intersects the rectangle being viewed on
  // the screen, we need to draw the part of the picture that intersects
  // with the screen.
  if ( SectRect( &pPictureInfo->rectPosition, &rectView, &rectSect ) )
  {
    // The intersection rectangle specifies the part of the screen that
    // contains the picture to be drawn.  To determine the part of the
    // picture that should be drawn to the screen, simply offset the
    // screen rectangle by the top left corner of the picture's location
    // within the text edit record.
    rectSectPicture = rectSect;
    OffsetRect( &rectSectPicture,
                -pPictureInfo->rectPosition.left,
                -pPictureInfo->rectPosition.top );

    // Convert the intersection rectangle into coordinates relative to the
    // screen containing the text edit record.  This is where on the screen
    // the picture will be drawn.
    OffsetRect( &rectSect, 0, -sCurrentHeight );

    // Copy the part of the picture that needs to be drawn on screen.
```

```
            CopyBits( &pPictureInfo->bitmapPicture,
                      &pInfoWindow->pDialog->portBits,
                      &rectSectPicture, &rectSect,
                      srcCopy, nil );
        }

    // Unlock the picture information handle since we're done with it.
    HUnlock( (Handle) pInfoWindow->hPictures );
}
```

CHAPTER 9

READING AND WRITING FILES

Most games will need to read and write files to disk. For example, any game that supports a save game feature will need to save information to disk about a game in progress. In addition, you may want to write high scores, help text, or other types of information to disk. You could use the standard I/O functions provided by the C language to support file I/O in your game, but they do not provide the flexibility and performance that the Macintosh toolbox provides. In addition, you will want to use the standard Macintosh File Open and Save As dialog boxes used to get file names from the user. These functions are supported by the File Manager, and are the topic of discussion for this chapter. This chapter covers those file manager routines that will be needed to support disk I/O for game programs. More advanced file I/O features such as those used to format disks or modify the file system are not covered.

Volumes

Before you start reading and writing files, you need to know where those files are located. The toolbox routines to create and open files take a volume reference number to specify in which disk and folder the file exists. A *volume reference number* uniquely identifies a specific folder on a particular drive. When the user chooses a file to save or load, you will be given the volume reference number of the drive and folder specified by the user. Associated with a volume reference number is the volume's name. The volume's name is fully *qualified*, meaning that each nested folder is contained in the name. The volume's name begins with the name of the drive followed by a colon, and then followed by a list of all the folders leading to the folder specified by the volume's name. Each folder is separated by a colon. For example, the volume name for a folder named **My Folder** contained within the **Game Folder** folder of the disk named **My Disk** would be **My Disk:Game Folder:My Folder**.

The Current Volume

The *current volume* specifies the current drive and directory and is used as a starting point when you display the standard File Open and Save As dialog boxes. In other words, the first folder's contents displayed to the user in a File Open or Save As dialog box is the current volume. The user, of course, can change the folder and disk using the standard File dialog boxes. Your game should change the current volume to the one specified by the user after they have completed a File Open or Save dialog box so that they will start out in that folder the next time a File dialog box is displayed. If you do not change the current volume, the user will be put back to the original current volume the next time a File Open or Save dialog box is displayed. That can be very annoying.

Which volume is current when your game first runs? The initial current volume is determined by the location your game. This will be the drive and folder containing you game program.

Getting and Setting the Current Volume

The following two toolbox routines allow you to get and set the current volume. Note that for `SetVol()`, you can set the current volume using either a fully qualified volume name or a volume reference number. The volume reference number is used if the volume name parameter is set to `nil`, otherwise the volume name parameter is used.

```
// Returns the current volume's name and reference number.
OSErr GetVol( StringPtr     pStringName,
              short         *psReferenceNumber );

// Sets the current volume, using either a volume name or
// volume reference number.  You specify nil for the name
// in order to set the current volume to the supplied
// reference number.
OSErr SetVol( StringPtr     pStringName,
              short         sReferenceNumber );
```

File Creator and File Type

A file's creator and type were discussed in the context of Finder icons back in Chapter 3 on resources. When it comes to creating files, you will need to specify a file creator and file type. Both are four character fields. Most of the time, the file creator of files your game creates should match the file creator of the game itself. This allows the Finder to associate that file with your game (and allows the Finder to display the custom icon you created for a file of that type). On occasion, however, you might want to specify a file creator other than your game's file creator. For example, if you create a text file that you want the user to look at using TeachText or SimpleText, you can specify a creator of 'ttxt', which is the creator type for TeachText and SimpleText. By doing so, TeachText or SimpleText will be started when the user double-clicks on the text file.

The file type field can be used to differentiate between the types of files your game supports. For example, Desert Trek saves both game files

and high scores lists. By using a different file type for each, the Finder displays a different icon for each type of file. In addition, the file type is used when your program displays the standard File Open dialog box. Only files of the type you specify are listed in the standard File Open dialog box. For example, you can specify to only have saved game files displayed when the user tries to open a saved game. This will prevent other types of files from being selected by the user.

The Standard File Dialog Boxes

The Macintosh toolbox provides two standard File dialog boxes that can be used by any program. These two dialog boxes are commonly referred to as the standard File Open dialog box and the standard File Save As dialog box. The standard File Open dialog box allows the user to specify a file to load. The standard Save As dialog box allows the user to specify a file name and location of a file to save. See Figure 9.1 for an example of these dialog boxes.

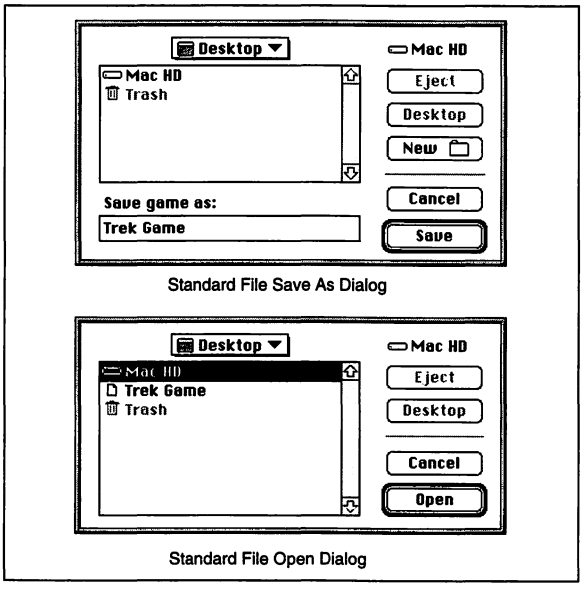


Figure 9.1 The standard Open and Save As dialog boxes.

The routines used to display the standard File dialog boxes return a standard file reply record. This record contains information related to the file specified by the user. The following describes the standard file reply record.

```
struct SFReply {
     Boolean   good;       // True if the user clicked OK
     Boolean   copy;       // Not used
     OSType    fType;      // File type
     short     vRefNum;    // Volume reference number
     short     version;    // File version
     Str63     fName;      // File name
};

typedef struct SFReply SFReply;
```

The `good` field specifies whether the user clicked on **OK** or **Cancel**. You should continue the open or save operation only when the value of this field is `true`. If it is `false`, you should cancel the open or save operation. The `fType` field specifies the type of the file selected by the user. The volume reference number returned will need to be used when creating or opening the file specified by the user. Remember, it is common practice to set the current volume to the volume specified by the user. That way, the next time a standard File dialog box is displayed, the first folder shown will be the one the user last specified. The file's version number is currently always set to **0** by the toolbox. The last field of the standard file reply record is the file's name. Though enough storage is allocated for file names of up to 63 characters, they are really limited to 31 characters. Remember that the file name specified here is a `pascal` string.

Displaying a Standard File Open Dialog Box

To display the standard Open dialog box, use the following toolbox routine. Note that the contents of the current volume will be initially displayed when this dialog box is drawn.

```
// The file type list used in SFGetFile().
typedef OSType SFTypeList[4];
```

```
// Displays and processes all events for the standard file
// open dialog.  It returns when the user dismisses the
// dialog.  This is a system modal dialog.
void SFGetFile(  Point          ptTopLeft,
                 Str255         str255Prompt,
                 ProcPtr        pProcFilter,
                 short          sNumberTypes,
                 SFTypeList     sfTypeList,
                 ProcPtr        pProcHook,
                 SFReply        *psfReply );
```

This routine takes a number of parameters, so let's take a look at them one at a time. The first parameter, `ptTopLeft`, specifies the top-left corner of the dialog box. This is where the dialog box will be displayed on the screen, meaning that the coordinates specified are global coordinates. Most of the time, you will want the dialog box centered nicely on the screen. To do so, simply specify a point of (0, 0). This will cause the toolbox to automatically center the dialog box on the screen for you. The next parameter, `str255Prompt`, specifies the text used to prompt the user. However, you'll notice that there is no prompt in the standard File Open dialog box. That means that this parameter goes unused. Maybe Apple will decide to use it in the future. The two `ProcPtr` parameters are used to customize the behavior of the standard File Open dialog box. They will not be discussed here since your game will rarely, if ever, need to use them in games. You can specify up to four file types to be displayed in the File Open dialog box. Use `sfTypeList` to specify the list itself, and `sNumberTypes` to specify how many types from that list should be used. If you specify a value of -1 for `sNumberTypes`, all file types will be displayed. Last, you need to specify a pointer to standard file reply record. If the user selects a file to open, its information will be returned to you in this record.

Displaying a Standard File Save As Dialog Box

To display a standard File Save As dialog box, use the following toolbox routine. Note that the contents of the current volume will be initially displayed when this dialog box is drawn. If the user specifies the name of a file that already exists, this toolbox routine automatically asks the user if they want to replace the existing file.

```
// Displays and processes all events for the standard file
// save as dialog box.  It returns when the user dismisses the
// dialog box.  This is a system modal dialog box.
void SFPutFile(   Point          ptTopLeft,
                  Str255         str255Prompt,
                  Str255         str255FileName,
                  ProcPtr        pProcHook,
                  SFReply        *psfReply );
```

This toolbox routine takes several of the same parameters as `SFGetFile()`. The top-left corner of the dialog box's location is specified as the first parameter. Most of the time, you will cause it to be centered on the screen by using a point coordinate of (0, 0). The prompt string is used in this dialog box and is displayed just above the text edit box that allows the user to type in the name of the file to be saved. The `str255FileName` parameter is used to set the initial contents of the text edit box where the user types in the file name. You will typically initialize this to be the name of the file if it has already been saved, or "Untitled" if this is the first time the file is being saved. The `ProcPtr` parameter is used to customize the Save As dialog box, and won't be discussed here. Last, the information of the file to be saved is returned to your program in the standard file reply record you specify.

An example on how to use `SFGetFile()` and `SFPutFile()` is shown at the end of this chapter.

Creating and Deleting Files

You need to create a file before it can be opened and read from or written to. In addition, you will occasionally need to delete files you've previously created. The toolbox provides routines to create and delete files for you. To create a file, use the following toolbox routine.

```
// Creates a new file.  If the file already exists, an error
// is returned (dupFNErr).
OSErr Create(   Str255       str255FileName,
                short        sVolumeReferenceNumber,
                OSType       osTypeCreator,
                OSType       osTypeFileType );
```

You need to specify the name of the file to create, the volume reference number of the drive and folder in which you want to create the file, the file's creator, and the file's type. The file's name and volume reference number will come from the appropriate fields in the standard file reply record. Your game will specify the file's creator and type. If the file you're trying to create already exists, you will get a `dupFNErr` return code (see the following file I/O errors section for a list of return codes). In this case, you can replace that file with the new one by deleting the file and creating it again. We'll see an example of this later.

To delete a file, use the following toolbox call.

```
// Deletes the specified file.
OSErr FSDelete(   Str255        str255FileName,
                  short         sVolumeReferenceNumber );
```

You can rename files using the following toolbox call. The renamed file resides in the same volume (disk and folder) as the original. If you specify a disk or folder name, that disk or folder gets renamed.

```
// Renames a file.
OSErr Rename(   Str255          str255OldFileName,
                short           sVolumeReferenceNumber,
                Str255          str255NewFileName );
```

Opening and Closing Files

Before reading and writing to a file, you need to open that file. Opening a file will return a file reference number, which you'll need when performing I/O operations to that file. After you have finished with a file, you must close it. The following two toolbox routines open and close files.

```
// Opens a file.  You need to specify the file name and
// volume reference number.  A file reference number is
// returned to your program.
OSErr FSOpen(   Str255      str255FileName,
                short       sVolumeReferenceNumber,
                short       *psFileReferenceNumber );
```

```
// Closes a file.
OSErr FSClose( short sFileReferenceNumber );
```

Examples of these calls appear later in this chapter.

Positioning the File Mark

A file's *mark position* determines exactly where in a file you read and write. In other words, bytes are written to or read from a file starting at the file's current mark position. After reading or writing, the file's mark position automatically changes to the location in the file just after the last byte read or written. You can obtain or change the file's mark position manually using the following toolbox calls. In most cases, you will not need to use these functions because you typically read or write through the entire file at one time, as opposed to moving through it randomly. For example, when saving a file, you almost always want the item you're about to write to be placed just after the last item you've written.

```
// Obtains a file's current mark position.
OSErr GetFPos( short    sFileReferenceNumber,
               long     *plMarkPosition );

// Sets a file's mark position relative to the base value
// you specify.  Use one of the base constants defined
// below.
OSErr SetFPos( short    sFileReferenceNumber,
               short    sMarkBase,
               long     lMarkOffset );

// File mark base constants used in SetFPos().
enum {
     fsAtMark = 0,       // Uses current mark and ignores offset.
     fsFromStart = 1,    // Relative to start of file.
     fsFromLEOF = 2,     // Relative to logical end of file.
     fsFromMark = 3      // Relative to current mark.
}
```

Note that you can use positive or negative values for the mark offset parameter in `SetFPos()`. If you try to set the file's mark position to a

point before the start of the file, you will get a `posErr` return code and the file's mark position will be moved to the start of the file. If you try to move a file's mark position to a point after the end of the file, you will get a `eofErr` return code and the file's mark position will be moved to the end of the file. Also note that if you use the `fsAtMark` base constant in `SetFPos()`, the offset parameter is ignored, meaning that the file's mark position remains unchanged.

If you want to obtain a file's logical end-of-file mark, you can use the following toolbox call. You can use this call to determine the size, in bytes, of an open file.

```
// Obtains a file's logical end-of-file position.
OSErr GetEOF(   short         sFileReferenceNumber,
                long          *plEndOfFile );
```

Reading and Writing Files

To read and write to files, you need to specify a pointer in memory where you want the data to go to or come from. Most of the time, this will be a pointer to one of your program variables. In addition, you need to specify the number of bytes to be read or written. If you are reading or writing a program variable, this length typically will be the size of that variable. Examples will be given toward the end of this chapter.

The following two toolbox calls read and write data to files. You will need the file's reference number to use these routines. When writing to a file, the end-of-file is automatically extended whenever the data you write exceeds the file's current end-of-file position. If you try to read past the end of a file, an error of `eofErr` is returned to your game. In this case, the number of bytes read stops at the end of the file. This means that you do not necessarily read the number of bytes you specified (this can also happen when writing to a file, as in the case of a disk full error). In cases of an error, only some of the bytes may be read or written. In other words, an error doesn't automatically mean that zero bytes are trans-

ferred. Sometimes you'll need to know how many bytes were actually read or written. This information is provided back to you in the same variable you used to specify how many bytes to read or write to the file. Thus, the value of the byte count variable you pass to these toolbox routines starts out as the number of bytes you want read or written to the file. After the routine completes, the value of the byte count variable you supplied is changed to the actual number of bytes read or written to the file. In cases where no error occurs, both values will be the same.

```
// Writes data to an open file.  The logical end-of-file is
// automatically extended if you are writing past the end of
// the file.
OSErr FSWrite(  short       sFileReferenceNumber,
                long        *plByteCount,
                Ptr         pOutput );

// Reads data from an open file.  eofErr is returned if you
// try to read past the end of the file.
OSErr FSRead(   short       sFileReferenceNumber,
                long        *plByteCount,
                Ptr         pInput );
```

TIP

There are times when an item you want to write to a file varies in length. For example, the size of the journal kept in Desert Trek depends on the number of moves and specific actions taken by the player. When saving a game, this doesn't really present a problem because Desert Trek can simply query the length of the text contained within the journal and write the appropriate number of bytes to the saved game file. However, when it comes time to load a saved game file, Desert Trek needs to know the length of the text contained within the journal saved to disk. An easy solution to this problem is to write out a length indicator just before the data of varying length. In other words, when saving a game, Desert Trek writes a length indicator just before the journal text. When the game is loaded, the length indicator is read first and specifies the number of bytes that must be read to obtain all of the journal's text. An example of this follows soon.

TIP

How about another tip when it comes to saving game files for your game? Often, your game will evolve over time as you make improvements and bug fixes. Frequently, this will cause the format of your game save file to gradually change. In other words, you might save more information to a game file in a later version of your game that adds new features. This could cause problems if a player tries to load an older save game file into the newer version of the game (or a newer save game file into an older version of your game). To prevent problems, the first thing you should write to a saved game file is a version number indicating the version of the saved game. When you change the format of the saved game file, increment the version number. Then, check this version number when loading a game. If it doesn't match the version number your game is looking for, you can take the appropriate action. You might simply display a message to the user stating that the saved game cannot be loaded by your game's version. Or, you might convert an older saved game file so that it can be used with your game's current version.

File I/O Errors

All file I/O routines return a result code that tells you whether or not the I/O completed successfully. If the I/O failed, the return code describes the type of failure. Your game needs to check for and process return codes so that an appropriate message can be displayed to the user if a file could not be saved or loaded. Why? You do not want the user thinking that a game was saved when it wasn't, otherwise they will not be too happy when they try to load the game at a later time. In addition, you do not want to load half a saved game, encounter an error, and try to act as if the entire saved game was loaded. Can you say impending system error?

The following is a list of most of the result codes that might be returned by the I/O routines discussed in this chapter.

```
enum {
        noErr = 0,              // No error
        dirFulErr = -33,        // Directory full
        dskFulErr = -34,        // Disk full
        nsvErr = -35,           // No such volume
        ioErr = -36,            // I/O error
        bdNamErr = -37,         // Bad name
        fnOpnErr = -38,         // File not open
        eofErr = -39,           // End of file
        posErr = -40,           // Positioned before start of file
        mFulErr = -41,          // Memory full
        tmfoErr = -42,          // Too many files open
        fnfErr = -43,           // File not found
        wPrErr = -44,           // Diskette is write protected
        fLckdErr = -45,         // File is locked
        vLckdErr = -46,         // Volume is locked
        fBsyErr = -47,          // File is busy
        dupFNErr = -48,         // Duplicate filename
        opWrErr = -49,          // File already open for writing
        paramErr = -50,         // Bad parameter
        rfNumErr = -51,         // Refnum error
        gfpErr = -52,           // Get file position error
        volOffLinErr = -53,     // Volume not on line error
        permErr = -54,          // Permissions error
        volOnLinErr = -55,      // Drive volume already on-line
        nsDrvErr = -56,         // No such drive
        noMacDskErr = -57,      // Not a Mac diskette
        extFSErr = -58,         // Volume belongs to an external fs
        fsRnErr = -59,          // File system internal error
        badMDBErr = -60,        // Dad master directory block
        wrPermErr = -61,        // Write permissions error
        dirNFErr = -120,        // Directory not found
}
```

TIP

If you encounter a file I/O error during the save of a file, you should delete that file because it's no good. Doing so will prevent the user from trying to load that bad file in the future. We'll see this tip in action in the save file coding example that follows.

You will need to check the return code of every file I/O call you make, so it's a good idea to create your own routine that does so. Desert Trek uses the following routine to determine if an I/O completed successfully. This function, which can be found in **File I/O.c**, returns true if the I/O was successful. If the I/O failed, this function calls a routine to display the I/O error to the user, and returns false. You can find the Desert Trek routine that displays an I/O error to the user, `PostIOError()`, in **File I/O.c**.

```
Boolean IOSuccessful( OSErr osErr )
{
  Boolean  bSuccess = true;

  // If the I/O error wasn't noErr, display an error message to the user.
  if ( osErr != noErr )
  {
    PostIOError( osErr );
    bSuccess = false;
  }

  // If there was no I/O error, return true.  Otherwise, return false.
  return( bSuccess );
}
```

Setting the Cursor

Reading and writing to disk can take some time, during which the player is typically prevented from interacting with your game. For example, the player is generally not allowed to enter moves while a saved game is being loaded. That simply wouldn't make much sense. In order to tell the user that they are waiting for a disk operation to complete before they can continue with the game, you need to change the cursor from the standard arrow to a watch.

The following data types are defined by the Macintosh for cursors. Basically, a cursor record contains the cursor image, a mask image, and the point that denotes the cursor's hot spot. Since a cursor is a 16 pixel by 16 pixel entity, it is the hot spot that specifies exactly the coordinate at which the mouse clicks occur.

```
struct Cursor {
   Bits16 data;
   Bits16 mask;
   Point hotSpot;
};

typedef struct Cursor Cursor;
typedef Cursor *CursPtr, **CursHandle;
```

Cursors are loaded from your game's resource fork. This means that you can use ResEdit to draw any cursor you want for use in your game. The following toolbox routine loads a cursor. Make sure to use `ReleaseResource()` to release the cursor once you are finished with it.

```
// Loads a cursor from the resource fork.  Use
// ReleaseResource() once you are finished with the cursor.
CursHandle GetCursor( short sCursorID );
```

There are several standard cursors defined by the Macintosh, and you load them just as you would any other cursor (see Figure 9.2). Use one of the following cursor IDs to load a standard cursor.

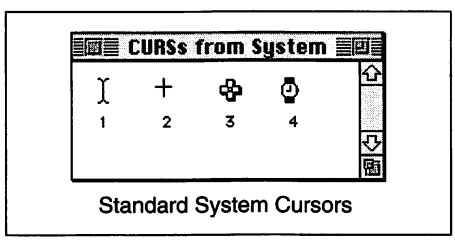


Figure 9.2 The standard cursors.

```
enum {
     iBeamCursor = 1,    // I-beam cursor.
     crossCursor = 2,    // Cross cursor.
     plusCursor = 3,     // Plus cursor.
     watchCursor = 4     // Watch cursor.
};
```

The following code snipped loads the watch cursor. The entire function can be found in **File I/O.c**.

```
static CursHandle hCursWatch;

void InitializeFileIO( void )
{
  hCursWatch = GetCursor( watchCursor );
}
```

To set the cursor, use the following toolbox routine.

```
// Sets the cursor.
void SetCursor( CursPtr pCursor );
```

If you want to set the cursor back to the standard arrow pointer, use the following toolbox routine.

```
// Sets the cursor back to the standard arrow.
void InitCursor( void );
```

Examples of these calls follow in the save example.

Save Example

The following routines save a Desert Trek game file. Basically, a Desert Trek saved game file contains three items: a *version number*, the *game's current state*, and the *journal*. Notice that the game's current state is defined by a large number of variables (distance traveled, time of day, inventory, health, etc.), but that these variables are defined in a single structure. This speeds disk I/O because all the game variables can be saved or loaded in a single I/O operation. If the game's variables were not all contained in a single structure, each variable would require its own disk I/O.

Before looking at the function that saves a Desert Trek game, let's look at two supporting functions. The first supporting function creates and opens a file, returning a file reference number. This function gets called whenever Desert Trek saves any type of file. If a file of the same

name already exists, that file is deleted and replaced with the new file. This function can be found in **File I/O.c**.

```
static short OpenNewFile( SFReply *psfReply,
                          OSType  osTypeCreator,
                          OSType  osTypeType )
{
  short sFile = 0;
  OSErr osErr;

  // Set the current volume to the volume of the new file.
  SetVol( nil, psfReply->vRefNum );

  // Create the new file.  The file creator and type are passed into this
  // routine.  The file name and volume are specified in psfReply.
  osErr = Create( psfReply->fName, psfReply->vRefNum, osTypeCreator,
                  osTypeType );

  // If this file already exists, delete it and create it again.
  if ( osErr == dupFNErr )
  {
    FSDelete( psfReply->fName, psfReply->vRefNum );
    Create( psfReply->fName, psfReply->vRefNum, osTypeCreator, osTypeType );
  }

  // Open the file.  The IOSuccessful() routine displays any error message to
  // the user.  IOSuccessful() returns true if no I/O error occurs.  If the
  // file was opened successfully, set the cursor to the watch.
  if ( IOSuccessful( FSOpen( psfReply->fName, psfReply->vRefNum, &sFile ) ) )
    SetCursor( *hCursWatch );

  // If the file wasn't opened, set the file reference number to 0.
  else
    sFile = 0;

  // Return the file reference number.
  return( sFile );
}
```

The second support function prompts the user for a file name. This function displays the standard File Save As dialog box and, if the user specifies a file to save, calls the previous routine to create and open the file. This function can be found in **File I/O.c**.

```
static short PromptForSaveAS( short   sPromptID,
                              short   sNameID,
                              Str255  str255NamePrompt,
                              OSType  osTypeCreator,
                              OSType  osTypeType,
                              SFReply *psfReply )
{
  Str255  str255Prompt;
  Str255  str255Name;
  short   sFile = 0;
  Point   ptOrigin = { 0, 0 };

  // Get the file save as prompt string.
  GetIndString( str255Prompt, FILE_STRINGS, sPromptID );

  // If no file name was specified, get the default file name string.
  if ( !str255NamePrompt )
    GetIndString( str255Name, FILE_STRINGS, sNameID );

  // If a file name was specified, copy it into the name prompt string.
  else
    memcpy( str255Name, str255NamePrompt, *str255NamePrompt + 1 );

  // Display the standard file save as dialog.
  SFPutFile( ptOrigin, str255Prompt, str255Name, nil, psfReply );

  // If the user chose to save a file, update the game windows (since they
  // were probably obscured by the standard file save as dialog) and open
  // the file.
  if ( psfReply->good )
  {
    UpdateWindows();
    sFile = OpenNewFile( psfReply, osTypeCreator, osTypeType );
  }

  // Return the file reference number.
  return( sFile );
}
```

The following function actually saves a Desert Trek game file. The user may or may not be prompted for a file name. If the user chose **Save As...** from the File menu, or the game has not yet been saved, the user is prompted for a file name. If the game has already been saved (and the

user did not choose **Save As...**), the game is saved over the old save game file for the current game. This function can be found in **File I/O.c**.

```
// Private variable that keeps track of whether or not the current game has
// been saved.
static Boolean bGameSaved = true;

// The current game's name.  It's "Untitled" if the game hasn't yet been
// saved.
static char    szSavedGameName[FILE_NAME_SIZE + 1];

// The current game's volume reference number.
static short   sSavedGameVolume;

Boolean  DoSave( Boolean bPromptForFileName )
{
  PGAME_STATE  pGameState;
  SFReply      sfReply;
  short        sFile = 0;
  long         lSize;
  long         lLongSize = sizeof( long );
  CharsHandle  hCharsJournal;

  // Default to a state of not saved (in case an I/O error occurs).
  bGameSaved = false;

  // If we need to prompt for a name...
  if ( bPromptForFileName ||
       !strcmp( szSavedGameName, (Cstr) *hStringUntitled ) )
  {
    // Convert the current game's name to a PASCAL string, since that's what
    // PromptForSaveAS() takes.
    CtoPstr( szSavedGameName );

    // Prompt the user for a file name.
    sFile = PromptForSaveAS( SAVE_GAME_STRING, 0, (Pstr) szSavedGameName,
                             'DTRK', 'TRKG', &sfReply );

    // Convert the saved game name back to a C string.
    PtoCstr( (Pstr) szSavedGameName );

    // If the file was opened, save the file's name and volume reference
    // number so that we can use them when the user saves the game again.
```

```
    if ( sFile )
    {
      sSavedGameVolume = sfReply.vRefNum;
      PtoCstr( sfReply.fName );
      strcpy( szSavedGameName, (Cstr) sfReply.fName );
    }
  }

  // If the user is not to be prompted for the file name of the game to save...
  else
  {
    // Set the volume reference number and name fields of the standard file
    // reply record to the current game's volume reference number and name.
    // It is the standard file reply record that gets used by the
    // OpenNewFile() routine to open the file.
    sfReply.vRefNum = sSavedGameVolume;
    strcpy( (Cstr) sfReply.fName, szSavedGameName );
    CtoPstr( (Cstr) sfReply.fName );

    // Open the saved game file.
    sFile = OpenNewFile( &sfReply, 'DTRK', 'TRKG' );
  }

  // If the saved game file was successfully opened...
  if ( sFile )
  {
    // Get the current game state.
    pGameState = RetrieveGameState();

    // Write the saved game version number to the file first.  To do so, first
    // compute the size of the version number (which is actually a string,
    // e.g. "1.01").  Then, write it to the file.  IOSuccessful() will display
    // any I/O error to the user.  If there is no I/O error, IOSuccessful()
    // will return true.
    lSize = sizeof( GAME_SAVE_VERSION );
    bGameSaved = IOSuccessful( FSWrite( sFile, &lSize, GAME_SAVE_VERSION ) );

    // If the last operation didn't generate an I/O error, write the game's
    // current state to the saved game file.  The entire structure is
    // written out using a single write operation.
    if ( bGameSaved )
    {
      lSize = sizeof( GAME_STATE );
      bGameSaved = IOSuccessful( FSWrite( sFile, &lSize, pGameState ) );
    }
```

```
    // Finally, if no I/O errors have occurred, write the game's journal to
    // the saved game file.
    if ( bGameSaved )
    {
      // First write the text length of the journal to the file.  We'll need
      // to know this when loading the saved game file.
      lSize = GetJournalText( &hCharsJournal );
      bGameSaved = IOSuccessful( FSWrite( sFile, &lLongSize, &lSize ) );

      // Finally, if there's no I/O error, write the text of the journal.
      if ( bGameSaved )
        bGameSaved = IOSuccessful( FSWrite( sFile, &lSize, *hCharsJournal ) );
    }

    // Close the file.
    FSClose( sFile );

    // If an I/O error occurred, delete the corrupt game file.
    if ( !bGameSaved )
    {
      // We need to convert the C string containing the file name to a
      // pascal string that can be used by the FSDelete() toolbox routine.
      CtoPstr( szSavedGameName );
      FSDelete( (Pstr) szSavedGameName, sSavedGameVolume );
      PtoCstr( (Pstr) szSavedGameName );
    }

    // Restore the cursor to the arrow (it was changed to the watch).
    InitCursor();
  }

  // Return true if the game was saved successfully.
  return( bGameSaved );
}
```

Loading Files Opened from the Finder

Before taking a look at an example of how to load a saved game file in Desert Trek, there's one last topic that needs to be discussed. The user might start your game by opening a saved game file from the Finder. In that case, the Finder automatically starts your game and passes it information concerning the file opened by the user. Your game should check

to see if it was started in such a manner, and if so, load the file opened by the user.

The information concerning those files that were opened in the Finder can be found in an application file record. The application file record contains the volume reference number and name of the file opened, as well as the file's type (in case your game supports more than one file type). There are two actions that the user can perform on your game files. The first is to open the file, and the second is to print the file. You can distinguish between these two actions using constants defined by the toolbox. If your game supports printing, you'll need to print the file specified.

```
// Constants to determine if the file was opened or printed
// from the finder.
enum {
     appOpen = 0,     // Opened
     appPrint = 1     // Printed
};

// The application file structure.
struct AppFile {
     short      vRefNum;       // File volume reference number.
     OSType     fType;         // File type.
     short      versNum;       // File version.
     Str255     fName          // File name.
};

typedef struct AppFile AppFile;
```

To determine if your game was started in response to a file being opened from the finder, use the following toolbox routine. Call this routine when your game first starts to see if you need to load a file opened by the user.

```
// Returns a message and file count.  The message is either
// appOpen or appPrint.  The file count is the number of
// files that were selected when open or print was selected
// from the Finder.
void CountAppFiles( short          *psMessage,
                    short          *psFileCount );
```

If the file count returned by `CountAppFiles()` is greater than zero, you know that your game was started in response to the user selecting **Print** or **Open** on one of your game's documents. In that case, you need to obtain information about the file or files selected. For most games, you will only concern yourself with one file because it usually doesn't make sense to open more than one saved game at a time. Also, many games will ignore the **Print** command. The following toolbox call returns file information for game documents selected when your game was started. You obtain file information one file at a time by specifying the index of the file on which you want information. The first file's index is one (not zero).

```
// Obtains information on a file that was selected when your
// game is started in response to a Finder open or print.
void GetAppFiles(  short       sIndex,
                   AppFile     *pAppFile );
```

After you have processed a file using GetAppFiles(), you need to tell the Finder that you have finished with that file. The following toolbox routines does just that.

```
// Clears an application file after you have processed it.
// Supply the index of the file you have processed (use the
// same index as you did in GetAppFiles()).
void ClrAppFiles( short sIndex );
```

Example

The following function from Desert Trek determines whether or not a saved game file needs to be opened when the game first starts. Because Desert Trek does not support printing, any documents opened due to a **Print** command from the Finder are ignored. Also note that only saved game files are processed, and even then only the first saved game file is processed (any others are ignored).

```
void CheckFinderOpenFile( void )
{
  short    sMessage;
  short    sNumberDocs;
```

```
    short    sLoop;
    AppFile  appFile;
    Boolean  bGameFound = false;

    // Count how many files were selected in the Finder when Desert Trek was
    // started.
    CountAppFiles( &sMessage, &sNumberDocs );

    // If any files were selected, and the message was not a print message,
    // search through all the files selected, looking for saved game files.
    if ( ( sMessage != appPrint ) &&
         ( sNumberDocs > 0 ) )
      for( sLoop = 1 ; sLoop <= sNumberDocs; sLoop++ )
      {
        // Get the file's information.
        GetAppFiles( sLoop, &appFile );

        // Tell the Finder that we've processed the file.
        ClrAppFiles( sLoop );

        // If the file is a Desert Trek saved game and we haven't already come
        // across a saved game, load that game.  If more than one saved game
        // file was selected, the first one we come across will be the one
        // loaded.
        if ( !bGameFound && ( appFile.fType == 'TRKG' ) )
        {
          bGameFound = true;
          ResumeGame( &appFile );
        }
      }
}
```

Load Example

The following two functions from Desert Trek load a saved game file. They can both be found in **File I/O.c**. The first function opens a saved game file. The name of the file to open is either obtained from the user via the standard File Open dialog box or from the Finder's application file record passed to Desert Trek when the user opens a saved game file from the Finder.

```
static short OpenSavedGame( AppFile *pappFile )
{
  Point      ptOrigin = { 0, 0 };
  OSErr      osErr;
  short      sFile = 0;
  SFReply    sfReply;
  // Only allow the user to select Desert Trek saved games from the standard
  // file open dialog.
  SFTypeList sfTypeList = { 'TRKG', '\p', '\p', '\p' };

  // If we're using the Finder information due to a user opening a saved game
  // file from the Finder, set up the standard file reply record as if the
  // user selected that file from the standard file open dialog.
  if ( pappFile )
  {
    sfReply.good = true;
    sfReply.vRefNum = pappFile->vRefNum;
    strcpy( szSavedGameName, PtoCstr( pappFile->fName ) );
  }

  // If the file was not opened by the Finder, display the standard file open
  // dialog to the user.
  else
  {
    // Display the standard file dialog.
    SFGetFile( ptOrigin, "\p", nil, 1, sfTypeList, nil, &sfReply );

    // If the user selected a file, save the name of the file.
    if ( sfReply.good )
    {
      PtoCstr( sfReply.fName );
      strcpy( szSavedGameName, (Cstr) sfReply.fName );
    }
  }

  // If we are to open a file...
  if ( sfReply.good )
  {
    // Save the file's volume reference number and set the current volume to
    // the file's volume.
    sSavedGameVolume = sfReply.vRefNum;
    SetVol( nil, sSavedGameVolume );
```

```
    // Convert the file name to a PASCAL string, since that's what FSOpen()
    // takes.
    CtoPstr( szSavedGameName );

    // If the file is opened without an error, set the cursor to the watch,
    // otherwise, set sFile to zero so that the calling routine knows that
    // the open failed.
    if ( IOSuccessful( FSOpen( (Pstr) szSavedGameName, sSavedGameVolume,
                               &sFile ) ) )
      SetCursor( *hCursWatch );

    else
      sFile = 0;

    // Convert the saved game file name back to a C string.
    PtoCstr( (Pstr) szSavedGameName );
  }

  // Return the saved game's file reference number.
  return( sFile );
}
```

The previous function is called by the main routine to load a saved game file. After opening the saved game file, this routine first checks to make sure that the saved game file isn't an old version. If so, an alert is displayed to the user and the file is not loaded. If the saved game file is current, the game state and journal are read from the file.

```
Boolean LoadGame( AppFile *pappFile )
{
  GAME_STATE  GameState;
  Boolean     bLoaded = false;
  short       sFile;
  long        lSize;
  char        szVersion[] = GAME_SAVE_VERSION;
  long        lLongSize = sizeof( long );
  Handle      hText;

  // Open the saved game file.  Either the user will be prompted with the
  // standard file open dialog, or the application file record supplied by
  // the Finder will be used.
  sFile = OpenSavedGame( pappFile );
```

```
// If the file opened successfully...
if ( sFile )
{
  // First load the saved game file version string.
  lSize = sizeof( szVersion );
  bLoaded = IOSuccessful( FSRead( sFile, &lSize, szVersion ) );

  // If the version string was loaded successfully, check it to see if it
  // matches the version string that this version of Desert Trek expect.  If
  // not, display an alert to the user and fail in loading the file.
  if ( bLoaded )
    if ( strcmp( GAME_SAVE_VERSION, szVersion ) )
    {
      ShowAlert( OLD_SAVE_VERSION, nil, nil, SetModalDialogMenuState );
      bLoaded = false;
    }

  // If the version string checked out okay, load the game state record.
  if ( bLoaded )
  {
    lSize = sizeof( GameState );
    bLoaded = IOSuccessful( FSRead( sFile, &lSize, &GameState ) );
  }

  // If the game state record loaded okay, load the size of the journal.
  if ( bLoaded )
    bLoaded = IOSuccessful( FSRead( sFile, &lLongSize, &lSize ) );

  // If the journal's size was loaded okay...
  if ( bLoaded )
  {
    // Allocate a new handle large enough to hold the journal.
    hText = NewHandle( lSize );

    // If the handle was allocated successfully...
    if ( hText )
    {
      // Lock the handle and load the journal text from disk into the
      // handle.
      HLock( hText );
      bLoaded = IOSuccessful( FSRead( sFile, &lSize, *hText ) );

      // If the journal text loaded okay, clear the journal and add the
      // saved game's journal text to the journal.
```

```
        if ( bLoaded )
        {
          ClearJournal( false );
          AddJournalText( *hText, lSize, true );
        }

        // Unlock the handle and free the memory associated with it.
        HUnlock( hText );
        DisposHandle( hText );
      }
    }

    // Close the file and set the cursor back to the standard arrow.
    FSClose( sFile );
    InitCursor();
  }

  // If the game loaded okay, set the game state to the one loaded from the
  // saved game file.
  if ( bLoaded )
    SetGameState( &GameState );

  // Return true if the game was loaded successfully.
  return( bLoaded );
}
```

Saving TeachText (SimpleText) Files with Embedded Graphics

You can save TeachText files with embedded graphics. In later versions of the Macintosh system software, TeachText has been replaced with SimpleText, so this trick also works for SimpleText. To add pictures to a TeachText file, you need to save the pictures into the resource fork of the TeachText file. The first picture must have a 'PICT' resource ID of 1000, the second 1001, and so on. To specify where in the text a picture is located, you need to write a "magic code" with the hexadecimal value of 0x00CA into the text part of the file. The first occurrence of this special value will cause the PICT resource of ID 1000 to be placed in the text. The second occurrence of the special value will cause the PICT resource

of ID 1001 to be placed within the text. Note that all pictures will be centered. Also make sure to leave enough blank lines in the text for the picture; otherwise, the picture will overlay some of the text. To see an example of how Desert Trek saves pictures into a TeachText file, see the `SaveInfoWindowText()` function in **Information Window.c**.

CHAPTER 10

INCORPORATING SOUND

No game is complete without sound. Sound effects and background music are almost mandatory if you want your game to be a success. The Macintosh sound manager provides a wealth of routines to support the playing and recording of sounds. This chapter covers those routines needed to support the playing of sound and music in your game. You will learn how to play digitized sounds asynchronously, support looping background music, and play multiple sounds simultaneously through multiple sound channels. Note that in order to use most of the routines described in this chapter, your game must be running on a Macintosh using system 6.07 or later.

Sound Formats

There are a plethora of sound formats in existence out there. Each format has its own advantages and disadvantages, and of course, each requires a different method to be played on the Macintosh. The remainder of this chapter focuses on the playing of the first two types of sound formats discussed below.

'snd ' Sound Resources

Probably the most popular and easiest to use sound format is the 'snd ' resource, also known as a sound resource. These types of sounds are also commonly referred to as System 7 sounds because they can be placed in the system folder and used as alert sounds. They can also be played directly in system 7 by double-clicking on a file containing a sound resource. The built-in sound recording capability provided by the Macintosh saves sounds in this format. To be sure, when it comes to sound effects and short music clips, this is the sound format of choice to use on the Macintosh.

As the name implies, sound resources are stored in the resource fork of a file or program as type 'snd '. This means that you can store as many sound effects as you need in the resource fork of your game, eliminating the need for a separate sound file. You load these resources using the generic `GetResource()` toolbox routine, and supply the resource handle to the sound manager routine that plays these types of sounds.

A sound resource is really a sampled, or digitized sound. When the sound is recorded, you can use any sampling rate up to 64 kHz, but the following rates are most commonly used: 22 kHz, 11 kHz, 7 kHz, and 5 kHz. The higher the sampling rate, the better the sound quality, however, you pay for that quality in terms of sound size. This means that the higher the sound quality, the larger the size of the sound. The sound manager provides a method to compress digitized sounds using the Macintosh Audio Compression and Expansion compression algorithm (MACE) and allows sounds to be compressed by 3:1 (MACE3) or 6:1 (MACE6). Compressed sounds take up considerably less space, but they

lose some sound quality and require a little more CPU time to play because they need to be decompressed on the fly.

AIFF and AIFF-C

For larger sounds such as music, and for sounds that you wish to be more portable across platforms or intended for use in other Macintosh programs, Apple and third-party developers have defined the Audio Interchange File Format (AIFF) and the Audio Interchange File Format extension for Compression (AIFF-C). Sounds and music recorded in AIFF and AIFF-C formats have their sound information stored in the data fork of a file. This overcomes any size limitations imposed by the maximum single resource size and makes the sound easier to share with other programs (it's easier to share a data file than it is to share a single resource contained in a resource file).

The sound contained within QuickTime movies is typically stored in this format. In fact, you can use SimpleText to convert a CD music track into an AIFF format music file, which can be played by any QuickTime player, or from within your game.

Other Sound Formats

Other popular sound formats include MOD files, MIDI files, and WAV files. MOD files are a very popular cross-platform format used to support multivoice music. MOD files tend to be much smaller than sound resources or AIFF files for a given playing time, but the playing of MOD files tends to be more difficult than using the built-in sound manager routines that play sound resources and AIFF files. MIDI files have similar properties to MOD files in terms of size and complexity to play. WAV files are pretty much the Microsoft Windows equivalent to a Macintosh sound resource (except that WAV files are not resources), meaning that they represent sampled, or digitized sound. You can't use WAV files directly with the sound manager routines, but tools exist to convert between the WAV file format and the sound resource format. Of course, many more sound formats exist, but they won't be discussed here.

Sound Channels

All sounds played in the Macintosh are played through *sound channels*. A sound channel is really just a queue that holds sound channel commands. This means that you can queue up multiple sound commands to a sound channel. Each command is processed by the queue in a first come, first serve basis, and one command completes before the next command begins. There are a number of different commands that can be sent to a sound channel, and not all of them necessarily produce sound. We'll see different types of sound commands throughout this chapter.

As with just about any other Macintosh construct, a sound channel has a record associated with it that contains information pertaining to that sound channel. Most of the fields contained within the sound channel record are used internally by the sound manager, but there's one user information field that can be used by your program for any purpose. We'll see a common use of this field later on in the section on playing music from disk. The following is the definition of a sound record, showing the user information field.

```
struct SndChannel {
     .
     .
     .
     long userInfo;
     .
     .
     .
};
typedef struct SndChannel SndChannel;
typedef SndChannel *SndChannelPtr;
```

Your game must create a sound channel before playing sound. To be honest, that's not technically true because you can have the sound manager create a sound channel for you when you want to play a sound. However, a game program would never do so because sound resources would be played synchronously, meaning that your game would grind to a complete halt until the sound finished playing. In addition, if you play music

from disk and have the sound manager automatically create a sound channel for you, you would have no further control over the playing of that sound file. In other words, you couldn't pause, resume, or stop the music in response to a game event.

The sound manager allows your game to create more than one sound channel so that you can have multiple sounds playing at the same time. In other words, you could have background music playing while at the same time play sound effects for certain events in your game. The number of sound channels that you can have open and playing sound simultaneously is limited only by the CPU power of the machine your game is running on. The more sound channels you have open, the higher the CPU load. Later on, we'll see methods to determine exactly how much CPU time the sound manager is taking to support your game's sound channels.

Creating Sound Channels

To create a sound channel, use the following sound manager routine.

```
// Creates a new sound channel.  A sound channel pointer is
// returned in ppSndChannel.  You can specify nil for
// pProcCallback if you don't need a callback function
// associated with this sound channel.
OSErr SndNewChannel(  SndChannelPtr          *ppSndChannel,
                      short                  sSynthesizerID,
                      long                   lInitParms,
                      SndCallBackProcPtr     pProcCallback );
```

The first parameter you need to pass to `SndNewChannel()` is a pointer to a sound channel record pointer. You will almost always pass a pointer to `nil` for this parameter, which causes the sound manager to automatically allocate memory for the sound channel for you. Note that you never pass `nil` itself as this parameter, but rather a sound channel pointer whose value is set to `nil`. We'll see an example later in this chapter.

The second parameter to `SndNewChannel()` is the *ID* of the synthesizer to use for the sound channel. There are currently three different types of synthesizers supported by the sound manager, defined as follows.

```
// Sound channel synthesizer IDs.
enum {
     squareWaveSynth = 1,     // Square wave synthesizer
     waveTableSynth = 3,      // Wave table synthesizer
     sampledSynth = 5,        // Sampled sound synthesizer
}
```

To play sound resources, AIFF files, and AIFF-C files, you need to specify a sampled sound synthesizer for this parameter. All routines and examples shown in this chapter assume that you are using a sound channel defined with a synthesizer ID of `sampledSynth`.

The third parameter to `SndChannelNew()` is the initialization parameter for the sound channel. The following initialization parameters can be used for sampled sound synthesizer channels.

```
// Sound channel initialization parameters.
enum {
     initChanLeft = 0x0002      // Left stereo channel
     initChanRight = 0x0003,    // Right stereo channel
     initMono = 0x0080,         // Monophonic channel
     initStereo = 0x00C0,       // Stereo channel
     initMACE3 = 0x0300,        // MACE 3:1
     initMACE6 = 0x0400,        // MACE 6:1
}
```

You can specify more than one initialization parameter by using addition or the bitwise OR operator. Typically you will choose one channel type parameter and perhaps one compression parameter. The `initChanLeft` and `initChanRight` initialization parameters specify that the sound channel should only play sound through the external audio jack on the Macintosh computer. The `initMono` and `initStereo` initialization parameters specify that the sound channel should play sound through both the external audio jack and the internal speaker. You will most likely choose from one of these two initialization parameters because you'll almost always want sound to be produced through the internal speaker in case the user doesn't have external speakers (if the user has external speakers hooked up to the audio jack, no sound will be produced through the internal speaker). The `initMACE3` and `initMACE6` parameters specify that the sound channel may have to play compressed

sounds. Sound channels can play uncompressed or compressed sounds regardless of the value of the compression initialization parameters, but these parameters provide the sound manager a way to better compute the CPU load required by the sound channel. Thus, you should specify a compression parameter if that channel is going to play compressed sounds. Doing so allows the sound manager to give your game a more accurate value of the CPU load used by that channel.

The final parameter to `SndNewChannel()` is a callback routine function pointer. This is a function you define that can be called by the sound manager when a particular command is processed by the sound channel. The most common use of the callback routine is to determine when sounds have finished playing. Callback functions will be discussed later in this chapter. If you do not need a callback function, you can specify `nil` for this parameter.

`SndNewChannel()` returns a sound manager error code if the sound channel could not be allocated. Return values include `noErr`, `resProblem`, and `badChannel`. See the section on sound manager errors for a description of these return codes.

Disposing a Sound Channel

After you have finished with a sound channel, you need to dispose of the channel in order to free up the resources taken by that channel. The following sound manager routine disposes of a sound channel.

```
// Closes and disposes of a sound channel.  If bQuiteNow is
// true, the channel closes immediately.  If false, queued
// commands are completed before the channel is closed.
OSErr SndDisposeChannel(  SndChannelPtr    pSndChannel,
                          Boolean          bQuietNow );
```

The first parameter to `SndDisposeChannel()` is a pointer to the sound channel to be disposed of. The second parameter specifies whether you want sound commands in the channel to be processed before the channel is closed. If this parameter's value is `true`, the sound channel's command queue is flushed, any sound currently playing is stopped, and

the channel is closed immediately. If this parameter's value is `false`, the sound channel does not close until all queued commands have been processed. You will typically specify `true` for this parameter because you'll want the channel closed right away (otherwise, why are you closing the channel?). Result codes for `SndDisposeChannel()` include `noErr` and `badChannel`.

Playing a Sound Resource

Before playing a sound resource, you need to load that resource from the resource fork of your game. In addition, you need to make sure that the sound resource does not move in memory while it is playing. This means that you must lock the sound resource before playing it. After the sound has finished, you can unlock it. Note that it is common practice to load all sound resources and lock them when a game initializes. This will speed up the playing of sounds during game play because they will not need to be loaded or locked when played. If you decide to use this strategy, make sure to move the sounds to the top of the heap before locking them (using the `MoveHHi()` toolbox routine) so that they do not create any heap compaction problems. Though Desert Trek loads all sound effect resources at game initialization, it does not lock them because it is not an arcade game and thus doesn't need every ounce of speed. The sounds are locked just before they are played, and unlocked after they have finished.

The following sound manager routine plays a sound resource.

```
// Plays a sound resource to the specified sound channel. A
// value of nil can be specified for the sound channel, in
// which case the sound is always played synchronously.  A
// value of true for bAsynch will cause the sound to be
// played asynchronously.
OSErr SndPlay(   SndChannelPtr    pSndChannel,
                 Handle           hSound,
                 Boolean          bAsynch );
```

The first parameter to `SndPlay()` is a pointer to the sound channel in which you want the sound to play. If you specify `nil` for this parameter, a sound channel is automatically created for you. In this case, the sound channel will be destroyed after the sound is complete. However, the sound will be played synchronously, meaning that your program grinds to a halt while the sound is playing. For this reason, games rarely use `nil` for the sound channel parameter.

The second parameter is the handle to the sound you want to play. Make sure that the sound is locked in memory so that it doesn't move while it is playing. The third parameter specifies whether the sound should be played synchronously or asynchronously. If `false`, the sound is played synchronously and control doesn't return to your program until the sound completes. Note that the whole system also comes to a grinding halt. This means that you should never play a sound synchronously. If you want your game to halt while the sound is playing, code it in such a way that you can still call `SystemTask()` to give other applications on the Macintosh processing time while the sound is playing. This means that you should always specify `true` for the last parameter so that the sound is played asynchronously.

`SndPlay()` can return one of the following error codes: `noErr`, `resProblem`, `badChannel`, or `badFormat`. Again, we'll see descriptions of these errors later in this chapter.

Quick Example

The following code snippet creates a sound channel, loads a sound resource, and plays it asynchronously through that channel. There's a lot more to managing sound than is shown in this example, but we'll take a closer look at that in a moment.

```
// Private global variables for sound management.
static Handle        hSound = nil;
static SndChannelPtr psndChannel = nil;
```

```
static void PlaySound( void )
{
  // Create a new sound channel which plays monophonic sound through the
  // internal speaker and audio jack, and which supports MACE 3:1 compression.
  // No callback function is supplied in this example for simplicity.  You'll
  // almost always want a callback function so that you'll know when a sound
  // completes.
  SndNewChannel( &psndChannel, sampledSynth, initMono | initMACE3, nil );

  // Load the sound resource, move it to the top of the heap, and lock it.
  hSound = GetResource( 'snd ', THE_SOUND_ID );
  MoveHHi( hSound );
  HLock( hSound );

  // Play the loaded sound through the newly created sound channel.
  SndPlay( psndChannel, hSound, true );
}
```

Playing Additional Sounds

After playing a sound on a sound channel, what happens next? That depends on what you want to do. For starters, the sound you played will take some amount of time to finish. During that time, you may need to play another sound. Maybe just after firing the ship's lasers, a large asteroid impacts with the player's ship. Before the laser firing sound completes, you need to play the ship destruction sound. If you simply used the `SndPlay()` routine to play the new sound to the same sound channel currently playing the laser firing sound, what would happen? Remember, the sound channel contains a queue, which holds all the commands sent to it. The `SndPlay()` routine really just sends a command to the sound channel to play a sound. This means that a new sound request will queue up behind the first one, resulting in the new sound being played after the first one finishes. In most cases, this is not what you want to happen. You want the new sound to be played immediately. It would be silly to have the ship explosion sound occur a little after the ship actually explodes on the screen.

What are your options? The first would be to create another channel to play the new sound. More to the point, you would probably set up two sound channels to play sounds, and alternate between them. However, if a

lot is happening in the game, you will still run into situations where all sound channels are busy playing sounds when a new one needs to be played. Eventually, you are going to have to decide what to do with the new sound. Most of the time, you will probably cut short one of the currently playing sounds and start the new sound immediately. Rarely will you simply not play the new sound, and most likely you would never want to delay playing the new sound until the current ones finish. To resolve this situation, however, requires that you have some additional information.

First, you need to know if a sound channel is busy playing a sound. We'll see how to do that in a section later describing how to obtain sound channel information. Second, if you decide to stop a sound currently in progress in favor of playing the new sound, you need a way to stop the current sound. Here you have two options. First, you could simply dispose of the sound channel currently playing the sound, then create a new channel to play the new sound. That's probably the easiest thing to do, but perhaps not the fastest. Desert Trek uses this technique because it's a strategy game and time isn't extremely critical. Even so, the disposal and creation of a sound channel occurs very quickly. Second, you could simply stop the current sound playing on the channel, and then play the new sound. In this case, you wouldn't need to dispose of and create a new channel. This can be accomplished by sending a command to the sound channel, and bypassing the queue so that the command gets executed immediately (you wouldn't want the command to wait on the queue for the sound currently playing to finish, now, would you?). This brings up a good question. How do you send commands to a sound channel, and what commands can you send?

Sending Commands to a Sound Channel

There are several commands that you may want to send to a sound channel. In the previous discussion, you might want to stop a sound that is currently playing on the channel so that you can start playing a different sound. Or, you might want to obtain information about a sound channel,

such as whether or not it is currently playing a sound (whether the sound channel is considered busy or not).

Commands are sent to a sound channel using a sound command record. A sound command record contains three fields: the command itself and two parameters. The meaning of the two parameters depends on the sound command itself. The following is the sound command structure definition.

```
struct SndCommand {
     unsigned short     cmd;
     short              param1;
     long               param2;
};

typedef struct SndCommand SndCommand;
```

There are several ways to send commands to a sound channel. The first is to put the sound command at the end of the sound channel's queue. Using this method allows you to place multiple commands on the queue and have them executed one after another. The second method allows you to force the sound channel to immediately execute the command. You would use this method to send commands that can't wait to be executed, such as the command to quiet a sound channel. Finally, there are a couple of sound channel status commands that are sent to the sound manager using yet another sound manager routine.

Placing Commands on a Sound Channel's Queue

Using the following sound manager routine, you can place sound commands on a sound channel's queue. The commands you place in the queue will be executed one at a time in the order you placed them there.

```
// Sends a command to the specified sound channel.  bNoWait
// specifies whether you want to wait until the queue has
// room for the command if it's currently full.
OSErr SndDoCommand( SndChannelPtr   pSndChannel,
                    SndCommand      *psndCommand,
                    Boolean         bNoWait );
```

The first parameter taken is a pointer to the sound channel to which you want to send the command. The second parameter is a pointer to the command record you want to send. The third parameter specifies whether or not you want to wait for space to become available on the queue if it happens to be full. The default queue size is 128 commands, so unless you're heavily loading up the queue, you need not worry too much about the queue becoming full. Most of the time you will specify `true` for the last parameter because you don't want your game coming to a screeching halt if the sound channel's queue becomes full. This routine returns `noErr`, `queueFull` (if the channel is full and you specified `true` for `bNoWait`), or `badChannel`.

Executing Sound Channel Commands Immediately

In some cases, you'll want to bypass the sound channel's queue and execute a sound command immediately. For example, if you want to stop the currently playing sound by sending it a **Quiet** command, you don't want that command to wait on the queue until the current sound finishes. This kind of defeats the purpose of sending the **Quiet** command in the first place. Thus, the sound manager provides the following routine to send commands directly to a sound channel.

```
// Sends a sound command directly to the sound channel,
// bypassing the channel's queue.
OSErr SndDoImmediate(   SndChannelPtr    pSndChannel,
                        SndCommand       *psndCommand );
```

This routine takes a pointer to a sound channel and command to be executed. Return codes can be `noErr` or `badChannel`.

Sound Commands

So, exactly what sound commands can you send to a sound channel? Good question. There are quite a few commands that can be sent to a sound channel; however, we'll just look at those that are more commonly used by game programs.

Command	Description
`callBackCmd`	The **Callback** command causes the callback function defined for the sound channel to be executed. This topic is discussed in great detail later in this chapter. The parameters for this command are application defined, meaning that you can use them in any way that you like.
`flushCmd`	The **Flush** command flushes the sound channel's queue. In other words, all queued commands are removed from the queue. You would normally send this command with `SndDoImmediate()`. The command parameters are not used, and thus are ignored. Note that this command only flushes the commands stored in the queue, and has no effect on the command currently being processed. This means that any sound being played will finish normally.
`loadCmd`	The CPU **Load** command allows you to obtain how much CPU time would be required for a channel with the given properties. You would use this command before creating an actual sound channel to determine the amount of the CPU that would be taken by a new channel if created. The CPU load is expressed as a percentage, where 100 means 100% of the CPU (something you'd probably want to avoid). You must use the `SndControl()` routine, which will be described in the next section, to send this command, because information will be returned to your program in the parameters of the sound command record. Parameter 1 should be 0 when you send the command. When the command finishes, parameter 1 will

contain the CPU load value. Parameter 2 must contain the sound channel's initialization parameters, such as `initMono`, `initMACE3`, etc.

Command	Description
`pauseCmd`	The **Pause** command causes a sound channel to stop processing commands in the queue. The parameters are unused and thus ignored.
`quietCmd`	The **Quiet** command stops the sound currently playing on the sound channel. You should send this command using `SndDoImmediate()`. The parameters are not used.
`resumeCmd`	The **Resume** command causes a sound channel to resume the processing of sound commands on the queue. You normally send this when you want to cancel a **Pause** command. The parameters are not used.
`totalLoadCmd`	The **Total Load** command allows you to determine the total CPU load taken by all existing sound channels in addition to a newly created sound channel with the initialization parameters you specify. In other words, this command works like the **Load** command except that the CPU load returned is not just the CPU load of the channel to be created, but the CPU load of that channel added to the current CPU load taken by all existing channels. Again, you must use the `SndControl()` routine discussed below to send this command. Parameter 1 should be 0 on input, and will contain the total CPU load on output. Parameter 2 should contain the sound channel initialization parameters to be used for the new sound channel.

`waitCmd`	The **Wait** command stops a sound channel from processing queued commands for the specified amount of time. Parameter 1 contains the length of the wait, expressed in half-milliseconds. In other words, a value of 2 means to wait 1 millisecond. Because parameter 1 is an unsigned short integer, you can wait up to 65535 half-milliseconds, or 32767.5 milliseconds, or 32.7675 seconds. Parameter 2 is unused.

Example

The following code example sends a quiet command to the specified sound channel, using the `SndDoImmediate()` routine. This code is not used in Desert Trek.

```
static void QuietChannel( SndChannelPtr psndChannel )
{
  SndCommand sndCommand;

  // If the sound channel exists...
  if ( psndChannel )
  {
    // The quietCmd doesn't use the parameter fields.
    sndCommand.cmd = quietCmd;
    sndCommand.param1 = 0;
    sndCommand.param2 = 0;

    // Execute the command immediately, without waiting at the end of the
    // sound channel's queue.
    SndDoImmediate( psndChannel, &sndCommand );
  }
}
```

Obtaining Sound Channel Information

The sound manager provides several routines that return information about sound channels and the sound manager itself to your game. We

have already seen that the `loadCmd` and `totalLoadCmd` sound channel commands return information concerning how much CPU load a new sound channel would take. In addition, you can obtain information about an existing sound channel as well as information concerning the status of the sound manager itself.

CPU Usage of a New Sound Channel

To send the `loadCmd` and `totalLoadCmd` sound channel commands, use the following sound manager routine.

```
// Sends a loadCmd or totalLoadCmd to the sound manager in
// order to determine the CPU load of a sound channel with
// the specified characteristics.
OSErr SndControl(  short          sSynthesizerID,
                   SndCommand     *psndCommand );
```

The first parameter specifies the synthesizer ID of the synthesize you plan to use for the sound channel. Use one of the same IDs you'd use for the `SndNewChannel()` command. Again, all the examples in this chapter use the `sampledSynth` synthesizer ID. The second parameter to `SndControl()` is a pointer to the **Sound** command you want to send. Remember that parameter 2 of the sound command record should contain the initialization parameters that you would use when creating the sound channel with `SndNewChannel()`. Parameter 1 of the sound command record will contain the CPU load when `SndControl()` returns. The only error code returned by this routine is `noErr`.

Obtaining Information for an Existing Sound Channel

You can obtain information for an existing sound channel. The information is returned to your game in a sound channel status record, which is defined as follows.

```
struct SCStatus {
     Fixed              scStartTime;
     Fixed              scEndTime;
     Fixed              scCurrentTime;
     Boolean            scChannelBusy;
     Boolean            scChannelDisposed;
     Boolean            scChannelPaused;
     Boolean            scUnused;
     unsigned long      scChannelAttributes;
     long               scCPULoad;
};

typedef struct SCStatus SCStatus;
typedef SCStatus *SCStatusPtr;
```

Field	Description
`scStartTime`	If a sound is currently playing on the channel (as indicated by a value of `true` for `scChannelBusy`), this value contains the starting time in seconds of a sound being played from disk (the next section describes how to play sounds from disk).

Field	Description
`scEndTime`	If a sound is playing on the channel (as indicated by a value of `true` for `scChannelBusy`), this value contains the end time in seconds of a sound being played from disk.
`scCurrentTime`	If a sound is playing on the channel (as indicated by a value of `true` for `scChannelBusy`), this value contains the current time in seconds of a sound being played from disk.

`scChannelBusy`	This field contains a value of `true` if the sound channel is currently producing sound; it is `false` if no sound is playing on the channel.
`scChannelDisposed`	This is a reserved field used internally by the sound manager.
`scChannelPaused`	This field contains a value of `true` if the sound channel is paused (in other words, not processing commands from the queue).
`scUnused`	This field is reserved for sound manager use.
`scChannelAttributes`	This field contains the current sound channel attributes. The format of this field is the same as the values specified in the sound channel initialization parameters when the sound channel was created.
`scCPULoad`	This field contains the CPU load being used by the sound channel.

To obtain the status record for an existing sound channel, use the following sound manager routine.

```
// Obtains information about the specified sound channel.
OSErr SndChannelStatus( SndChannelPtr    psndChannel,
                        short            sLength,
                        SCStatusPtr      pscStatus );
```

You need to pass three parameters to this routine. First, you need to supply a pointer to the sound channel for which you want to gather information. Second, you need to supply the size of the buffer you're providing to gather the information. You will almost always specify the size of the sound channel status record. Finally, you need to specify a pointer to the sound channel status record that will receive the information. This routine can return error codes of `noErr`, `paramErr`, or `badChannel`.

Obtaining Information about the Sound Manager

You can obtain information concerning all sound channels that the sound manager is currently managing for all applications running on the Macintosh. Note that this includes sound channels allocated by other programs. The information you can obtain is defined by the following sound manager status record.

```
struct SMStatus {
     short       smMaxCPULoad;
     short       smNumChannels;
     short       smCurCPULoad;
};

typedef struct SMStatus SMStatus;
typedef SMStatus *SMStatusPtr;
```

Field	Description
`smMaxCPULoad`	This value is the maximum CPU load that the sound manager will not exceed when allocating sound channels. In other words, if you try to create a new sound channel that would increase the CPU load for all channels beyond this value, the sound manager will fail to create the new channel. The default value for this field is 100 when the system boots.
`smNumChannels`	This value contains the total number of sound channels allocated by all applications running on the Macintosh. Note that the channels do not necessarily have to be currently in use, just that they have been allocated.
`smCurCPULoad`	This value contains the current CPU load taken by all sound channels allocated on the Macintosh.

To obtain the sound manager information, use the following routine.

```
// Obtains information about all sound channels allocated by
// the sound manager on this Macintosh.
OSErr SndManagerStatus( short           sLength,
                        SMStatusPtr     psmStatus );
```

You need to specify the length of the buffer you're providing, typically the size of the sound manager status record. In addition, you need to specify a pointer to the sound manager status record that will receive the status information. This routine only returns `noErr`.

Example

The following code snippet makes calls to gather information on a specific sound channel as well as the sound manager as a whole.

```
static void GetSoundChannelInfo( SndChannelPtr psndChannel )
{
  OSErr    osErr;
  SCStatus scStatus;
  SMStatus smStatus;

  // Obtain information for the specified sound channel.
  osErr = SndChannelStatus( psndChannel , sizeof( scStatus ), &scStatus );

  // Obtain information on the sound manager as a whole
  osErr = SndManagerStatus( sizeof( smStatus ), &smStatus );
}
```

Playing Sound from Disk

Playing sounds effects with the `SndPlay()` sound manager routine works great for short sounds and music clips. However, what if you want to play a longer music clip, one that's very large in size (perhaps several megabytes)? The `SndPlay()` routine requires that the sound to be played be loaded and locked in memory, making it a poor choice for larger music clips. You can easily imagine situations where there isn't

enough memory available to load the entire music clip to be played. What's needed is a way to play a large music clip from disk, where portions of the music are loaded into memory as they are needed, and removed from memory when they have been played.

This is accomplished by providing a double buffer mechanism from which to play the music. In other words, two buffers are used to play the music, where each buffer is much smaller than the size of the entire music file. First, buffer one is filled with as much music as can be fit from disk. Next, while the music in buffer one is being played, the second buffer is loaded with the next part of the music from disk. Once the music in the first buffer is complete, the music in the second buffer is played, without causing a break in the music. Then, while the music in the second buffer is being played, the first buffer's contents are replaced with the next portion of music from disk. This process repeats until the entire music file has been played. The advantage of the double buffer scheme is that a music file of unlimited size can be played using a relatively small amount of memory. As long as the buffers are large enough to hold enough music so that while one is being played the other can be completely filled from disk, the entire music file plays without interruption. Fortunately, the management of the double buffer scheme is automatically taken care of for you when you use the play from disk sound manager routines.

Starting a Play from Disk Sound

The following sound manager play-from-disk routine allows you to play either a sound resource or an AIFF file from disk.

```
// Plays a sound resource or AIFF file from disk, using the
// double buffering technique to play continuous sound using
// a limited amount of memory.
OSErr SndStartFilePlay( SndChannelPtr    psndChannel,
                        short            sFileReferenceNumber,
                        short            sResourceID,
                        long             lBufferSize,
                        Ptr              pBuffer,
                        AudioSelectPtr   paudioSelection,
                        ProcPtr          pprocCompletion,
                        Boolean          bAsynch );
```

The first parameter to `SndStartFilePlay()` is a pointer to the sound channel that you want the sound file to be played. You could specify `nil` here, and have the sound manager automatically create a sound channel for you to play the music file. However, I strongly recommend against doing so because you will have no further control over that sound channel. You will not have the pointer to the sound channel that allows you to send it additional commands, pause the music, or stop the music. Therefore, you will almost always want to have some control over the sound from disk channel, even if it's to simply stop the music in response to the user turning off sound in your game.

The second and third parameters specify whether you want to play a sound resource or an AIFF (or AIFF-C) file. Remember, a sound resource is contained in a resource of type 'snd ', and an AIFF file contains sound information stored in the data fork of a file. If you want to play an AIFF file, supply the file reference number in `sFileReferenceNumber`, and specify a value of 0 for the resource parameter `sResourceID`. If you are playing an AIFF file, you must open the file first with the `FSOpen()` toolbox call, which will return to you the file reference number. If you want to play a sound resource, specify **0** for the file reference number, and the 'snd ' sound resource ID in `sResourceID`.

The fourth and fifth parameters deal with the double-buffering to be used to play the file. The total size of the double-buffer is specified in `lBufferSize`. Most of the time, you will specify `nil` for `pBuffer`, which will cause the sound manager to automatically allocate the double-buffers for you. In that case, the sound manager creates two buffers, each with a size half of the buffer size you specified in `lBufferSize`. If you want to allocate your own buffer, allocate one buffer of the size you specified, lock it so that it's fixed in memory, and supply a pointer to that buffer in `pBuffer`.

The sixth parameter, `paudioSelection`, allows you to specify that only a portion, or selection of the sound file be played. Details won't be provided here, other than you supply `nil` for this parameter to play the entire music file. The seventh parameter specifies a callback function to be called when the sound completes. This is a function you define that

takes some action once the music file completes. However, you can accomplish the same thing by defining a callback function for the sound channel you specified and sending that channel a `callBack` command. We'll see how to do that in the next section. For most circumstances, you will supply `nil` for this parameter too (and use the method described in the next section on callback routines to take action once the sound file is finished playing). If you do want to use a callback function for `SndStartFilePlay()`, note that it differs from a regular sound channel callback function in that the only parameter passed to the callback function is the sound channel pointer. After reading the following section on callback routines, you'll realize that you need to set the user information field of the sound channel in order to pass useful information to the callback routine (such as the A5 world pointer for your game, which we'll discuss in a moment).

The final parameter to this function specifies whether or not the music file should be played asynchronously. You will always, want to specify `true` here so that the music file gets played asynchronously. If you do not, your game and the entire Macintosh system running your game will grind to a halt until the music file plays completely.

This routine can return one of the following error codes: `noErr`, `notEnoughHardwareErr`, `queueFull`, `badChannel`, `badFormat`, `notEnoughBufferSpace`, `badFileFormat`, `channelBusy`, `bufferTooSmall`, or `siInvalidCompression`. Again, see the section on sound manager errors for a complete description of these return codes.

Example

The following code snippet starts playing an AIFF sound file from disk. This routine can be seen in its entirety in **Digitized Sound.c,** which includes the code to ensure that the sound channel's callback routine gets called once the sound file completes playing.

```
// Private global variable containing the file reference number of the music
// file being played from disk.
static short sMusicFile;
```

```
void PlayBackgroundSound( void )
{
  // The volume reference number where the AIFF sound file can be found.
  short sVolume;

  // If the background sound channel has not yet been created, create the
  // the sound channel.
  if ( !psndChannelBack )
    SndNewChannel( &psndChannelBack, sampledSynth, initMono,
                   (SndCallBackProcPtr) CheckSoundDone );

  // If the background sound channel has already been created, stop any music
  // currently being played on that channel.
  else
    CloseBackgroundMusicFile();

  // Obtain the volume reference number where the Desert Trek program lies.
  // That's where we're going to look for the music file.
  sVolume = GetProgramVolume();

  // Open the music file, and if successful, play the music on the background
  // sound channel.  Notice that a double buffer of 65536 bytes is being used.
  if ( !FSOpen( MUSIC_FILE, sVolume, &sFileMusic ) )
    SndStartFilePlay( psndChannelBack, sFileMusic, 0, 65536, nil, nil,
                      nil, true );
}
```

Pausing a Play from Disk Sound

Your game may want to pause the music being played from disk, so that it can be resumed from that point later on. The following sound manager routine will pause or resume a play from disk sound on a sound channel.

```
// Pauses or resumes sound on a channel playing sound from
// disk.
OSErr SndPauseFilePlay( SndChannelPtr psndChannel );
```

You simply pass a pointer to the sound channel currently playing sound from disk. If the channel is playing sound, it gets paused by this routine. If the channel is already paused, this routine causes sound to resume on that channel. Note that this routine only works for a channel playing

sound from disk. Return codes include `noErr`, `queueFull`, `badChannel`, and `channelNotBusy`.

Stopping a Play from Disk Sound

If you want to stop a sound being played from disk, use the following sound manager routine.

```
// Stops sound from a channel playing sound from disk.
OSErr SndStopFilePlay(  SndChannelPtr  psndChannel,
                        Boolean        bStopNow );
```

You need to pass a pointer to the sound channel currently playing sound from disk, and a parameter indicating whether or not you want the sound to stop immediately. You will always specify `true` for `bStopNow` because if you do not, this call doesn't return until the sound file completes, causing your game and the Macintosh on which it's running to grind to a halt. This routine can return `noErr` or `badChannel`. Don't forget to close the file if you are playing an AIFF sound file after the sound stops.

Example

The following routine from **Digitized Sound.c** stops a sound playing from disk, and closes the sound file.

```
// Private global variable containing the file reference number of the music
// file being played from disk.
static short sMusicFile;

static void CloseBackgroundMusicFile( void )
{
  // If the music file is open...
  if ( sFileMusic )
  {
    // Stop playing the music file on the background sound channel.
    SndStopFilePlay( psndChannelBack, TRUE );
```

```
        // Close the music file.
        FSClose( sFileMusic );

        // Set the music file reference number to 0 so that we know that the file
        // is now closed.
        sFileMusic = 0;
    }
}
```

Callback Routines

Callback routines for sound channels are commonly used when you want to take some action after a sound has finished playing. The action taken varies depending on your game's needs, but commonly you'll want to play another sound or close the sound channel once it has stopped playing a sound. For example, if you are playing background music by looping a short music clip, you need to play that clip again immediately after it's finished in order to simulate continuous sound.

A few technicalities need to be considered when using a callback routine for a sound channel. These technicalities arise due to the fact that your callback routine might be called at interrupt time. A routine being executed at interrupt time cannot allocate or free any memory, as calls to the memory manager aren't allowed. This means that you must be very careful what you do in your callback routine. Remember that even if you do not call a memory manager routine directly, a call to a different toolbox manager routine might in turn result in a call to the memory manager. In fact, it's best to simply set a global variable in the callback routine that gets checked in your game's main event loop. When your game's main event loop detects that the global variable has been set, you can take the action you really want to perform when the callback routine was invoked. However, you don't have direct access to your game's global variables when your callback routine gets called. This is due to the fact that the A5 world is not set to your game's global variables section in memory at interrupt time. Wait a second, what is this A5 world? Glad you asked.

The A5 World

The A5 world really refers to the application's globals that can be referenced at a particular time. It is called the A5 world because register A5 on the 680X0-based Macintoshes points to the global variables area of a program. In Chapter 2, we saw that a Macintosh application breaks up the memory allocated to it into three separate sections: the globals area, the heap, and the stack. The globals area contains an application's global variables, and when you reference those variables, it is via an offset from the A5 pointer. During interrupt time, the A5 world won't be set to point to your game's global variables. This means that your interrupt routine can't immediately access your game's variables. What you need to do is to set the A5 world to point to the global variables of your game. This can be accomplished by using the following toolbox routine.

```
// Sets the A5 world. This routine returns the current A5
// world so that you can set it back when done.
long SetA5( long lA5 );
```

The `SetA5()` routine sets the A5 world and returns the A5 world that was in force before you changed it. This means that you can set the A5 world to your game's A5 world in the callback routine to get access to your global variables. Just before your callback routine ends, you should set the A5 world back to the one you received when you changed it. How do you know the value of your game's A5 world pointer? The following routine returns the A5 value for your game.

```
// Returns the A5 world pointer for your application.
long SetCurrentA5( void );
```

When you set up the sound channel command that causes your callback routine to be called, you need to pass along the A5 value for your game in the second parameter of the sound channel command. You can then use this value in the callback routine to set the A5 world to point to your game's global variables. You will learn how to do this in the example given in the section on setting up a callback routine.

Callback Routine Definition

Your callback routine needs to be defined as follows.

```
// Function prototype for your game's sound channel callback
// routine.  Note that there is no toolbox call called
// MyCallback().
pascal void MyCallback( SndChannelPtr       psndChannel,
                        SndCommand          sndCommand );
```

Basically, your callback routine will be passed two parameters. This first is a pointer to the sound channel that caused the callback routine to be called. The second is the **Callback** command that triggered the callback routine. Note that this is not a pointer to the **Sound** command, but the actual record itself.

Setting Up a Callback Routine

You need to do two things to set up a callback routine so that it gets called when a sound finishes playing on a sound channel. First, you need to pass a pointer to your callback routine when creating the sound channel with `SndNewChannel()`. Second, you need to place the **Sound** command that invokes your callback routine on the sound channel's queue. This command should come after the **Sound** command that plays the sound. Remember that using `SndPlay()` really just causes a **Sound** command to be placed on the sound channel's queue. So, make sure that you place the **Callback** command after calling `SndPlay()` (or `SndStartFilePlay()` if you're playing music from disk).

As previously discussed, when your callback routine gets invoked, you will be passed the sound channel pointer and **Sound** command, which cause the callback routine to be called. This means that you can use the sound command parameters to pass information to your callback routine. One of these parameters should be the A5 world of your game so that you can access your game's global variables during the callback routine. Because the A5 world pointer is a long integer, you must use sound command parameter 2 because sound command parameter 1 is a short integer.

This leaves sound command parameter 1 for your own use. Desert Trek uses it to tell the callback routine which sound channel has completed playing a sound (either the background sound channel or the sound effects sound channel).

Example

The following function from **Digitized Sound.c** plays a sound effect, which sets up the callback function to be called when the sound finishes playing.

```
// Private global variable that points to the sound effect we want to play.
// The PlaySound() routine sets this variable based on what sound effect is
// to be played.  You can find the PlaySound() routine in "Digitized
// Sound.c".
static Handle hSound;

// This is the global variable that gets set in the callback routine when a
// sound effect finishes playing.  It's value gets checked during every pass
// of the main event loop.
static Boolean bForegroundSoundDone = FALSE;

static void PlayForegroundSound( void )
{
  SndCommand sndCommand;

  // If the sound channel does not yet exist, create it.  Note that we are
  // defining the Desert Trek function, CheckSoundDone(), as the callback
  // function for this sound channel.
  if ( !psndChannel )
    SndNewChannel( &psndChannel, sampledSynth, initMono | initMACE3,
                   (SndCallBackProcPtr) CheckSoundDone );

  // Move the sound to the top of the heap and lock it.
  MoveHHi( hSound );
  HLock( hSound );

  // Reset the flag used to check if the sound effect has finished.
  bForegroundSoundDone = false;

  // Set up the callback sound command.  The first parameter specifies that
```

```
  // the foreground sound channel caused the callback routine to be called.
  // The second parameter is set to Desert Trek's A5 world.
  sndCommand.cmd = callBackCmd;
  sndCommand.param1 = FOREGROUND_SOUND_COMPLETE;
  sndCommand.param2 = SetCurrentA5();

  // Play the sound effect.
  SndPlay( psndChannel, hSound, true );

  // Send the command to call the callback routine when the sound finishes
  // playing.
  SndDoCommand( psndChannel, &sndCommand, true );
}
```

Callback Routine Processing

When the callback routine gets called, you need to take some action. Again, the callback routine itself should just set a flag that causes some processing to get kicked off from your main event loop because you do not want to do anything in the callback routine that causes a memory manager routine to be called. The first thing your callback routine needs to do, however, is to set the A5 world to your game's A5 world. You can then feel free to access your game's global variables. The last thing your callback routine should do is restore the A5 world back to what is was before your callback routine was called. The following is Desert Trek's callback routine, which can be found in **Digitized Sound.c.**

```
static pascal void CheckSoundDone( SndChannelPtr  pSndChannel,
                                   SndCommand     sndCommand )
{
  long lA5;

  // Set the A5 world to Desert Trek's A5 world, which was stored in the
  // second parameter of the sound command.
  lA5 = SetA5( sndCommand.param2 );

  // If the first parameter indicates that this callback was invoked due the
  // finishing of sound on the background music sound channel, set the flag
  // which denotes that the background sound is done.
  if ( sndCommand.param1 == BACKGROUND_SOUND_COMPLETE )
    bBackgroundSoundDone = true;
```

```
  // If the first parameter indicates that this callback was invoked due the
  // finishing of sound on the foreground sound effects sound channel, set the
  // flag which denotes that the foreground sound is done.
  else if ( sndCommand.param1 == FOREGROUND_SOUND_COMPLETE )
    bForegroundSoundDone = true;

  // Restore the A5 world to what it was before this routine was called.
  SetA5( lA5 );
}
```

Desert Trek's main event loop calls the following routine to check if one of the sound channel done flags has been set (you can see the call made from the `CheckEvents()` function shown in Chapter 2 in the section, *Waiting for and Getting Events*). This routine can be found in **Digitized Sound.c.**

```
void CheckSound( void )
{
  // If the foreground sound effect is done, call a routine to destroy the
  // sound effects channel.
  if ( bForegroundSoundDone )
    KillForegroundSound();

  // If the background music sound is done, call a routine to play that sound
  // again in order to simulating continuous music.
  if ( bBackgroundSoundDone )
    PlayBackgroundSound();
}
```

Sound Manager Errors

Calls to sound manager routines can return the following errors.

Error	Description
`noErr`	No error occurred. The sound manager routine completed successfully.

`resProblem`	An error occurred while loading the resource.
`badChannel`	The sound channel is invalid, corrupt, or unusable.
`badFormat`	The 'snd ' resource you're trying to play is corrupt or unusable.
`queueFull`	The sound channel queue is full.
`paramErr`	A parameter to a sound manager routine is incorrect.
`notEnoughHardwareErr`	The current Macintosh is not capable of playing a sound from disk due to insufficient hardware.
`notEnoughBufferSpace`	There isn't enough memory available to allocate the buffer size specified in `SndStartFilePlay()`.
`badFileFormat`	The play-from-disk file specified is not a valid AIFF or AIFF-C file.
`channelBusy`	The **Play-from-Disk** command failed because the specified sound channel is already busy playing a sound from disk.
`buffersTooSmall`	The buffer size specified `SndStartFilePlay()` is not large enough to support play from disk.
`siInvalidCompression`	The file specified in `SndStartFilePlay()` has an invalid compression type.

Error	Description
`channelNotBusy`	Returned by `SndPauseFilePlay()` when the specified sound channel is not currently playing a sound from disk.

Suspend and Resume Events

Playing sound asynchronously brings up an interesting problem when you consider that the user can, at any time, switch out of your game and into another application (even if it's just the Finder). What exactly happens to the sounds that are currently playing on your sound channels once your game becomes inactive? The answer is that the sound manager will continue to play those sounds to completion. This behavior, though, doesn't really become a good Macintosh application. A good Macintosh program should stop all sounds in progress if the user switches to another application.

When your game receives a suspend event, you should close all sound channels, or at the very least, quiet them. When your game receives a resume event, you can reopen the sound channels and start playing any sounds, such as background music. Suspend and resume events are passed to your game in an event of type `osEvent`. However, you need to make sure your game is set up to receive these events. In other words, your game doesn't get these events unless you specify that your application accepts them. You specify that your game should receive suspend and resume events in the 'SIZE' resource and in the project type definition of your game project. If you are using Think C or Symantec C++, you must set the SIZE flags from the Set Project Type dialog box (from under the Project menu). Simply make sure that the Suspend & Resume Events flag is selected. If you are editing the 'SIZE' resource directly, you need to make sure that the Accept Suspend Events and Does Activate on FG Switch flags are set to **1**. These two steps are very important since if you do not complete them, your game will not receive suspend and resume events.

Getting a Suspend or Resume Event

When your game receives an operating system event, you need to determine if it was a suspend or resume event. Actually, both event types are grouped together into what the toolbox calls a `suspendResumeMessage`. This message type differentiates between suspend/resume messages and other types of operating system messages (which we will not discuss here). To determine whether the operating sys-

tem event received by your game is a `suspendResumeMessage`, you need to check the high byte of the `message` field of the event record. If the `message` field high order byte equals `suspendResumeMessage`, then you have a suspend or resume message. We'll soon see an example.

After determining that a suspend or resume message occurred, you need to figure out whether the event was a suspend event or a resume event. This is accomplished by checking bit **0** of the `message` field. If bit **0** is set, the message is a resume message. If the bit is reset, the message is a suspend message. The toolbox defines a constant, `activateFlag`, which can be used to determine the value of bit 0 of the `message` field.

The following code example, from **Main.c** handles operating system events.

```
static void HandleOSEvent( EventRecord *pEvent )
{
  WindowPtr pWindow;
  long      lEventType;

  // Get the high order byte of the event message field.
  lEventType = pEvent->message >> 24;

  // If the high order byte is equal to suspendResumeMessage, this is a
  // suspend or resume message.
  if ( lEventType == suspendResumeMessage )
  {
    // We need to activate or deactivate the frontmost Desert Trek window,
    // depending on the event type.
    pWindow = FrontWindow();

    // If this is a resume message...
    if ( pEvent->message & resumeFlag )
    {
      // Initialize the cursor back to the arrow pointer.
      InitCursor();

      // Activate the frontmost window (which really just enables the
      // scrollbar of that window, if it has one).
      ActivateDeactivateWindow( pWindow, true );

      // Start playing the background music (if it's turned on).
      PlayBackgroundSound();
    }
```

```
        // Otherwise this is a suspend message...
        else
        {
          // Stop all sound channels and dispose of them.
          KillBackgroundSound();
          KillForegroundSound();

          // Deactivate the frontmost window (which really just disables the
          // scrollbar control of that window, if it has one).
          ActivateDeactivateWindow( pWindow, false );
        }
      }
    }
```

Background Music Example

Desert Trek can play background music from two sources. First, included with the game is a short sound clip (stored as a 'snd ' resource) that can be looped to simulate continuous music. However, after repeated play, this sound clip could become repetitive. To overcome this problem, Desert Trek can play any AIFF or AIFF-C music file that the user wants. All the user has to do is put an AIFF or AIFF-C file named "Music" in the same folder as Desert Trek, and Desert Trek will play that file as the background music. This section will show the strategy and code used to play the background music in Desert Trek.

A separate sound channel is used to play the background music so that sound effects played on another channel don't interrupt the background music. This sound channel gets created when the program starts if the background music option is turned on. The channel remains open for as long as the music plays, meaning that if the music is stopped for some reason, the channel is destroyed. The music can be stopped either by turning off the background music option or switching to another application. If the music stops because the sound finishes playing, the background music sound is played again.

Because we want the background music to play uninterrupted, there are a few things we need to be careful about. If the short sound clip is used as the background music, the sound must be looped fairly often

(about every 15 seconds). If some lengthy processing is happening at the time the sound needs to be replayed, a pause in the music might occur. In fact, heavy processing isn't necessary to cause a pause in the music. If the user clicks on a menu and holds the mouse button down for a few seconds, thinking about what to do, control doesn't return to Desert Trek's main event loop until the user releases the mouse button. If the sound clip finishes while the user is thinking about what menu item to select, Desert Trek won't get a chance to replay the sound until the user finally makes a choice. Now, the chances that the user will hold the mouse button down for a long period of time just when the music clip finished isn't all that great, but it can happen.

In order to lessen the chances that such an action would cause a pause in the background music, Desert Trek puts an additional **Play** command on the sound channel's queue. Thus, if the first music clip completes while something else is going on, the music won't stop. Of course, if the user keeps the mouse button down long enough in a menu, the music will eventually stop. However, the user would have to hold down the mouse for quite some time in order for that to happen. When Desert Trek regains control, it places another **Play** command on the sound channel's queue. In practice, then, there should always be one additional **Play** command on the queue, giving Desert Trek a little extra time to process an end of music notification. This extra **Play** command is not used when playing sound from disk because it really can't be implemented. The **Play-from-Disk** command cannot be queued up because you would just receive a "channel busy" error code. However, this is less of a problem for the longer background music files, since they will probably be entire songs. If there's a short delay between when the song finishes and starts up again, it won't sound that bad.

Desert Trek's main event loop calls the `CheckSound()` routine to see if the background music sound finished playing. Because the event loop should be executed every 1/60th of a second, there will be no noticeable delay or pause in the background music (except in the previous case, but we've now taken care of that). However, if some internal Desert trek processing takes some time, we need to call the `CheckSound()` routine periodically during that processing to make sure that sound end notifica-

tions are handled. The only real "lengthy" processing that Desert Trek performs is the view transition special effect (which takes about half a second). The `NiceDelay()` function is used by the special effects routine to ensure that the effect is timed correctly regardless of the speed of the Macintosh. If you look in the `NiceDelay()` function, you'll notice that it calls `CheckSound()` to make sure that sound processing takes place while Desert Trek waits for something to finish.

The following routine starts the background music. It gets called when the program begins, when the user turns on background music (after it had been off), or when Desert Trek receives a resume event. The private global variables used to support background music are also shown.

```
// Used to determine that the background music sound is complete.
#define BACKGROUND_SOUND_COMPLETE 1

// The name of the music file that the user can supply.
#define MUSIC_FILE        "\pMusic"

// Determines whether background music is on or off.  It is saved in the
// resource fork of Desert Trek so that it can be read the next time Desert
// Trek is run.
static Boolean        **hbBackgroundSoundOn = nil;

// The background music sound channel pointer.
static SndChannelPtr  psndChannelBack = nil;

// Used for the background music clip's 'snd ' resource.
static Handle         hSoundBackground = nil;

// Set to true by the callback function when the background music sound
// completes.
static Boolean        bBackgroundSoundDone = false;

// The background music file's file reference number.  It is set to 0 when the
// file is closed or not being used.
static short          sFileMusic = 0;

void PlayBackgroundSound( void )
{
  SndCommand sndCommand;
  Boolean    bDoubleUp = false;
  short      sVolume;
```

```
// If the background music option is turned on...
if ( **hbBackgroundSoundOn )
{
  // If the sound channel hasn't been created yet...
  if ( !psndChannelBack )
  {
    // We want to place an extra sound clip on the sound channel's queue
    // so that if Desert Trek can't immediately process the end of music
    // notification, the background music doesn't stop.
    bDoubleUp = true;

    // Create the background music sound channel.  We'll allow stereo
    // output so that AIFF and AIFF-C files get played in stereo if the
    // user has external speakers attached to the audio jack.  The
    // CheckSoundDone() callback routine is defined for this sound channel.
    SndNewChannel( &psndChannelBack, sampledSynth, initStereo | initMACE3,
                   (SndCallBackProcPtr) CheckSoundDone );
  }

  // If the sound channel is open, that means that we are playing the sound
  // again since it has finished playing.  If the sound was an AIFF or
  // AIFF-C music file, we need to close that file before we try to play
  // it again.
  else
    CloseBackgroundMusicFile();

  // Reset the flag that tells Desert Trek that the background music has
  // completed.  This flag will get set to true by the callback routine
  // when the sound completes.
  bBackgroundSoundDone = false;

  // Set up the sound command that will cause the callback routine to be
  // invoked.  The command must be sent after the command which starts
  // playing the background music.  Note that we set the first parameter to
  // BACKGROUND_SOUND_COMPLETE to denote that the background music channel
  // caused the callback routine to be called.  The second parameter is set
  // to Desert Trek's A5 global variables pointer.  This is done so that the
  // callback routine can access Desert Trek's global variables.
  sndCommand.cmd = callBackCmd;
  sndCommand.param1 = BACKGROUND_SOUND_COMPLETE;
  sndCommand.param2 = SetCurrentA5();

  // Get Desert Trek's volume reference number so that we can use it to see
  // if the user has placed an AIFF or AIFF-C music file in the same folder
  // as the Desert Trek program.
  sVolume = GetProgramVolume();
```

```
    // If the music file exists and can be opened without error, start sound
    // play from disk.  We specify the music file's reference number and a
    // buffer size of 64K.
    if ( !FSOpen( MUSIC_FILE, sVolume, &sFileMusic ) )
      SndStartFilePlay( psndChannelBack, sFileMusic, 0, 65536, nil, nil,
                        nil, true );

    // If no music file was found, play the short music clip loaded from
    // Desert Trek's resource fork.
    else if ( hSoundBackground )
    {
      // Move the sound resource to the top of the heap and lock it before
      // playing it.
      MoveHHi( hSoundBackground );
      HLock( hSoundBackground );
      SndPlay( psndChannelBack, hSoundBackground, true );
    }

    // Send the sound command that will cause the callback routine to be
    // invoked.  Note that this command will be processed by the sound
    // channel after the sound we just played finishes.
    SndDoCommand( psndChannelBack, &sndCommand, true );

    // If we just created the sound channel and we are playing the short sound
    // clip from the resource fork, we want to queue up a second sound play
    // and callback command.  This will prevent a pause in the background
    // music if Desert Trek can't replay the music clip right away.
    if ( bDoubleUp && !sFileMusic && hSoundBackground )
    {
      SndPlay( psndChannelBack, hSoundBackground, true );
      SndDoCommand( psndChannelBack, &sndCommand, true );
    }
  }
}
```

When the background music sound completes, the callback function gets called by the sound manager. That callback routine sets the `bBackgroundSoundDone` flag, which in turn is detected by the `CheckSound()` routine. The `CheckSound()` routine will call the `PlayBackgroundSound()` routine when it detects that the `bBackgroundSoundDone` flag has been set. The callback routine, `CheckSoundDone()`, and the `CheckSound()` routine both have been

shown already in the section on callback routines presented earlier on in this chapter.

When the background music is to be stopped, any open sound file is closed and the sound channel is disposed of. The following routine handles the disposing of the sound channel itself and can be found in **Digitized Sound.c.**

```
void KillBackgroundSound( void )
{
  // If the background music channel has been created...
  if ( psndChannelBack )
  {
    // Close the background music file (if it's being used).
    CloseBackgroundMusicFile();

    // Close the sound channel, which automatically stops any sound playing
    // on that channel.
    SndDisposeChannel( psndChannelBack, true );

    // Set the background sound channel pointer to nil so that we know that
    // channel has been disposed and is not valid.
    psndChannelBack = nil;
  }

  // If the background music clip from the resource file has been loaded,
  // we can unlock it's handle since the sound is no longer playing.
  if ( hSoundBackground )
    HUnlock( hSoundBackground );
}
```

The following routine from **Digitized Sound.c** closes an AIFF or AIFF-C file being used for the background music.

```
static void CloseBackgroundMusicFile( void )
{
  // If the music file is open and being used...
  if ( sFileMusic )
  {
    // Stop the play from file command immediately.
    SndStopFilePlay( psndChannelBack, TRUE );
```

```
        // Close the music file.
        FSClose( sFileMusic );

        // Set the music file reference number to 0 so that we know that the
        // music file is no longer being used.
        sFileMusic = 0;
    }
}
```

CHAPTER 11

AFTER THE GAME IS FINISHED

Now that you've finished writing, what's going to be the hottest game in existence, what do you do next? Can you just sit back, relax, and collect royalties? Hardly. There's still a lot left to do if you want your game to be a great success. What you do with the game after you've finished writing it can have as much effect on the popularity and success of that game as the actual game itself.

First of all, when exactly do you consider a game complete? Finishing your game will most likely be the hardest part of writing that game. Even after you consider the game completely finished, there's the question of testing it out to make sure it works the way you intended, ensure that there aren't any bugs, and absolutely, positively make sure that it doesn't crash on any Macintosh systems you intend to support. After testing the game, you need to make it available to the world. There are numerous avenues of distribution, and choosing the right ones will go a long way to meeting your goals for the game. This is especially true if you plan to release your game as shareware, a popular form of distribution for games that aren't written on contract for a major game company. Last, how should you support your game after it has been released? Most of the time, writing and releasing a game is just the first step in a game's life time. Bug fixes and enhancements can dramatically alter a game over time.

This appendix will discuss these preceding questions, as well as provide recommendations on how to market, distribute, and support your game. Keep in mind that writing a game is only part of a game's life cycle. If you want your game to be really successful, you need to put in a lot of additional work.

Finishing Your Game

The hardest part of writing a game is finishing the last 10%. When you've reached this point, a bulk of the hard work is complete and you're most likely really anxious to release the game to the world and see the reactions it generates. You will be really tempted to skimp on those finishing touches that make a good game a great game. Don't succumb to this temptation. Take a little extra time and put in those finishing touches. You'll be well rewarded for the extra time spent at this stage of a game's life cycle.

What constitutes the finishing touches on a game? Really, just about anything that isn't directly related to game play itself can be considered a finishing touch. Typical features include integrated on-line help, introduction screens, additional sounds and music, high scores lists, and user interface shortcuts such as smart icons.

On-line Help

Most games include some sort of on-line help, even if it's just an abbreviated form of the game's playing instructions. On-line help provides a real benefit to users because they won't need to spend a lot of time reading instructions before playing a game. Most people want to start playing right away, and won't have the patience to read a complex readme file or printed instruction manual. For shareware games especially, you want to make sure that the game's rules are presented in a concise and easy-to-understand manner because the user won't give your game much play time if they can't quickly and easily figure out how to play. There are exceptions to this rule, especially for more complex simulations, but even then you'll want to provide on-line help so that users can refer to it while they learn how to play your game.

On-line help ranges from the very simple dialog box screen with a few instructions, to a very complex help system that provides detailed instructions, balloon help, and context sensitive hints, tips, and techniques. What you provide depends on the amount of time you wish to spend, and whether or not you intend to provide an external help file. The advantage to including all game instructions as on-line help is that there isn't an external help file that might get lost or deleted. However, may people like to print out the instructions for more complex games. For this reason, you may want to provide a print capability or the ability to save the on-line help as a text file, which can then be printed with TeachText or SimpleText. Desert Trek provides a good example for a decent middle-of-the-road help system. It provides styled text accessible by topic, embedded graphics for examples, and the ability to save help text into a text file. Also, the Help dialog box can be viewed at the same time the game is being played. However, it falls short of balloon help and context sensitive hints.

Introduction Screens

Many games provide an introduction screen or even a short "movie" to introduce the game. Introduction screens can definitely give a game a professional look, but can take a considerable amount of work on your

part to write depending on its complexity. A fairly easy introduction screen consists simply of a graphic contained within a window, with perhaps a few buttons that can be used to start a game, view its help, or set some options. A more complex introduction screen includes animated graphics, and perhaps a scrolling story line. In addition, an introduction screen might show some aspects of game play, usually by having a computer-controlled player actually playing parts of the game.

Being a simple game, Desert Trek provides no introduction screen. However, Galactic Empire, a shareware game that can be found on the CD-ROM included with this book, contains an introduction screen with a scrolling background story. Many commercial games contain lengthy and sometimes very impressive introduction screens, which the developers surely spent a lot of time working on. Keep in mind, however, that the user will only be interested in the introduction screen the first several times they play your game. After that, they'll most likely want to skip the introduction screens and get right to game play.

Additional Sounds and Music

Nearly all games need to provide sound effects in order to become popular. Most silent games won't hold a user's interest for long. There are exceptions to this rule depending on the type of game you've written, but rare is the game without any sound that becomes really popular. In addition to sound effects, background music can add a lot to a game. You should consider this when finishing your game. However, don't go overboard. Professional quality games do not contain an overabundance of sound, and the sounds they do contain are usually tasteful and add to the ambiance of the game itself. Keep in mind that additional sound and music adds to the size of your game. In fact, a large percentage of a game's size can usually be accounted for by its sounds and music. For shareware games especially, you need to be conscious of the size of your game because many people will be downloading the game from the Internet or an on-line service. The larger the game, the longer the download time.

High Scores Lists

Most games provide some type of high scores list. Again, they can range from a simple dialog box displaying the top 10 scores to more complex screens that contain game statistics as well as scores for multiple skill levels. Many games incorporate the high scores list into the game's introduction screen. What route you take depends on the type of game your write and the amount of time you are willing to spend designing and writing the high scores screen.

User Interface Enhancements

There are any number of things that you can do to improve the way in which the user interacts with your game. This includes features such as smart icons, the ability to undo commands, the ability to pause a game, and the ability to zoom, scroll, or position game windows. Many of these types of enhancements only make sense for certain types of games. For example, a pause feature really has no meaning for a game like Desert Trek. However, some enhancements can make any game easier to play. Your goal should be to make it as easy as possible for the user to play your game. They should be able to concentrate on the actual game play itself, not on how to enter moves, keep track of the game's status, or how to save or start a new game. Time spent working on user-interface enhancements can really go a long way to make a game successful. There have been examples of fun-to-play games that haven't become very popular because they had awful user interfaces. This is especially true for games that have been ported to the Macintosh from the DOS world, where the developers didn't take advantage of what the Macintosh has to offer in the area of user interface.

The best way to determine the types of user interface enhancements that would be good for your game is to look at what other successful games have done. Pay close attention to those aspects that make the game easy to play or hard to play. Feel free to be creative in your user interface, but don't deviate so far from the standard as to make it difficult for the

user to understand your interface. The user should be able to pick up on your basic interface without needing to read any instructions. It's okay to require a little learning for some shortcuts or more advanced interface features, but no user is going to read pages of documentation to figure out how to launch a fleet, trade commodities, or save a game in progress.

Testing Your Game

Now that you've finished writing your game, including adding all those little enhancements discussed above that will make your game great, what's next? Before you distribute your game to the world, make sure it works exactly the way you intended. You need to test your game.

Completely testing your game isn't going to be easy. First, make sure that the game doesn't crash on a wide range of Macintoshes, including a variety of system versions using various extensions. Second, make sure that your game behaves as you intended. For example, are all puzzles solvable, and is the game winnable? You also want to make sure that the game isn't too easy to win, otherwise players will loose interest quickly. Last, it is common to get feedback on your game from a select group of users at this stage. Their input will help you make improvements to your game.

Testing a game is commonly called *beta-testing*. Beta-testing means that your game is almost finished, and you're testing to make sure all features of the game work correctly. If you are still writing portions of the game, but need to test those parts that have already been written, you perform what's commonly called an *alpha-test*. Alpha-testing takes place while portions of the game are still being written. You will do most of the alpha-testing yourself as you write the game. You will need the help of others to beta-test your game.

Finding Testers

You are going to need help testing your game for several reasons. First and most obvious, you need to try out your game on different Macintosh platforms. Unless you own a wide range of Macintosh products (Powerbooks, Performas, Power Macs, and older Macintoshes such as

Quadras, Centrises, and Mac IIs), you need the assistance of others who own those types of machines. Second, other people will have different playing styles than your own. You may test your game to death, but someone with a different playing style may come across problems in your game that you never caught. Third, determine what others think of your game. Do they find it fun to play? Is it easy to learn? Last, you need to make sure your game isn't too easy or too hard to win. You yourself can't be the judge of that. Remember, you wrote the game and know all the tricks and solutions to all the puzzles.

The best way to find testers is to look for friends with Macintoshes. They will probably be more than happy to help you out. In addition, you can seek testers by posting articles to an Internet newsgroup (such as the `comp.sys.mac.games.*` hierarchy, or `comp.sys.mac.programmer.games`) or an on-line service such as America Online. You'll stand a good chance at getting a number of interested testers if you solicit such a large audience. You can also seek testers from user's groups.

Setting the Ground Rules

You need to be selective when it comes to choosing who will beta-test your game. This is especially true if you don't personally know the individual. You need to make sure that beta-testers are dedicated to actually testing your program. Just playing the game isn't going to be enough. They will need to stress the game and use all its features. More importantly, though, you need to make sure that they will report back to you their results in a timely fashion and continue testing newer versions of your game as you make bug fixes and modifications.

You should limit the number of people beta-testing your game; if you have too many, it will become difficult corresponding with all of them. Conversely, you'll probably want more than one or two testers to make sure you get a variety of feedback. It all depends on the quality of your testers. A small number of good testers will be much better than a larger quantity of so-so testers.

You should offer your beta-testers some incentive for testing your game. This incentive can range from something simple such as having

their name included in an acknowledgments screen somewhere in your game, to getting a free copy of the game once it's released. Also keep track of the types of computers and systems versions your prospective testers are using have. You don't want to end up with 10 testers using the same Macintosh and system version. Last, consider how you are going to send your game to the testers. If you want to use electronic methods, make sure that your testers can handle it. Keep in mind that you'll be making a number of changes during the testing phase, meaning that you'll need to send out new versions of your games to the testers from time to time.

Spell out your timeframe to your testers. If you want to release the game soon, you'll need testers who can spend a lot of time with your game now. Also make sure that your testers are dedicated to testing your game. You don't need people who just want a sneak preview of your game before it gets released. They won't provide much useful feedback. Also make it clear that the testers are not to distribute the beta-version of your game to others. The last thing you want is a beta-version of your game out on the net somewhere.

Getting Feedback

You should really listen to what your testers have to say, even if you aren't pleased by it. Keep in mind that of course you love your game, but others may have a different opinion. Remember that it is these types of people who will be paying for your game, so you need to make sure that they are going to like it. In addition to telling you how much fun your game is to play, testers will provide invaluable input concerning your game's user interface. They'll tell you if it makes the game easy to play, or just simply gets in the way.

Just as your testers are dedicated to testing your game in a timely fashion, make sure you fix bugs and make improvements in a timely fashion. You do not want the testing phase of game development to continue forever. However, don't send out a new version of the game every day. This can be quite expensive for you and your testers (if they have to pay

to download the game). Plan on having to send out two to five new versions of your game during testing.

Distributing and Marketing Your Game

Distributing and marketing your game means getting your game into the hands of game players, and maybe even receiving some type of compensation for your efforts. One of the most important decisions you'll have to make is whether to distribute your game as shareware, or try to make it into a commercial game. That decision will determine how and where you'll distribute your game, and the type and amount of compensation you'll receive. It is often easier to create and support shareware game than a commercial game, but you will almost always get more money for a commercial game. We'll see why in the following sections.

Commercial Distribution

If you are planning on writing a commercial game, you should have a contract up front with the company that's going to distribute your game. That contract will spell out the terms of your agreement with that company, and set forth how and how much you will get paid. A traditional contract gives all rights of the game to the company with which you are working. That means that they determine what to do with the game now and into the future. You are basically giving that company the right to do whatever they see fit with your game. In return, you will typically receive royalties on the sales of that game. The royalties are based on what the company sells the game to distributors for, not the retail price of the game. For example, a $20 retail price for a game might mean that the company selling it gets around $12. The royalties you receive will typically range from 10% to 15%, but recently, that number has been declining somewhat due to the fierce competition in the market. You'll have to share those royalties if your game is included in a multigame package. For example, if your game is included with four others, each game author would receive one-fifth of the royalties, or 2% to 3% (don't you just love

it when the math works out so easily in these examples!). In addition to royalties, you may also receive an advance against those royalties. An advance, typically a small sum ranging from as little as $500 to as much as $2,000 is exactly that: an advance. This means that enough royalties need to come in to cover that advance before you see any additional money. However, you do get the advance up front, before the game is complete.

Though it seems like you are giving up a lot to the company distributing your game, consider the compensation and alternatives. First, you will almost certainly sell more copies of a game if it is commercially distributed. This is due to the advertising capabilities of larger companies, and the fact that they can get your game on store's shelves and into mail-order catalogs. The company you're working with will also provide the packaging for your game, including artwork for the box. They will be able to mass produce your game in a cost-effective way, something that's difficult for an individual to accomplish. In a sense, all you have to do is write the game, and the company you're working with will take care of the rest.

However, also consider your responsibilities. If you sign a contract with a company for a game, you must deliver. Not only that, but your game must be of the highest quality and be really, really bug free. You also have a deadline, which means you must be very serious about writing a game. If it's a hobby or part-time job of yours, make sure you can spare the time needed to finish the game on time. Also make sure that you have the drive and experience to complete the game.

Getting a Contract

So, how do you go about getting a contract to write a game? Good question. To be honest, it's not easy. It helps to build a reputation for yourself in the Macintosh gaming community. Writing shareware games is one of the best ways to build a reputation for yourself. Many quality shareware games have gone on to become commercial games. In those cases, the company may contact you first, or you may submit a copy of your game to the company to see if they're interested in making a commercial version. Basically, it helps to have something concrete to show the company you want to commercially distribute your game, even if it's just a demo.

I can't really prescribe a sure-fire way to get you a contract. You mainly need to build a reputation for yourself. Do a good job at this, and you might be surprised at the opportunities that arise. You need to write high-quality games and get good visibility for those games. The section on shareware distribution and marketing will give you many good ideas on how to get started. After doing so, contact a couple of companies that distribute Macintosh games and see if they'd be interested in distributing a commercial version of one of your games. If they are of high quality, you stand a good chance at success.

Shareware Distribution and Marketing

Many games written by nonprofessionals are distributed as *shareware*. Shareware games are games that are distributed without immediate payment to the author. After playing the game, the user may choose to send some type of payment to the game's author in order to register that game.

Distributing a Shareware Game

An important aspect of releasing a shareware game is getting it into the hands of users. The more people who try your game, the more people who will register it. This means that you need to spend some time sending copies of your game to various places, both electronically and by mail. Where should you send your games to get the biggest exposure? You'll need to send copies of your games to the Internet, major on-line services, and major users groups. In addition, you may want to send your shareware games to major shareware distributors, though they typically obtain most of the shareware they distribute from the Internet.

So, where on the Internet should you send your game? There are a large number of ftp sites where Internet users download shareware programs. If you had to figure out where each site was located, and send your game to each of those sites, you'd probably get a really big headache. Fortunately, there's a central location that you can send your game and have it automatically distributed to the major shareware repositories. Send your game to `macgifts@mac.archive.umich.edu`. Send the

game in .hqx format (binhex format, which is a textual representation of a Macintosh application) by inserting it into an e-mail message. You should type a short description of the game just before where the binhex data for your game starts in that message. Programs such as StuffIt Light and Compact Pro allow you to generate binhex versions of your shareware game. Make sure to compress your game first, and do not make it a self-extracting archive. The goal here is to make your game as small as possible so as to reduce download time.

You should also send your game to the major on-line services such as America Online and CompuServe. You'll need to have an account or know someone with an account to upload your game to these on-line services.

There are three major users groups that you should seriously consider sending your shareware game to. These are the National Home and School Macintosh Users Group (NHSMUG), the Arizona Macintosh Users Group (AMUG), and the Berkeley Macintosh Users Group (BMUG). These users groups have worldwide recognition, a large number of members, and each offers CD-ROM collections containing shareware games. You will certainly get a lot of exposure for your games by getting them into the shareware collections of these users groups. To obtain information on AMUG, visit their web site `www.amug.org`. To obtain information on BMUG, visit their Web site at `www.bmug.org`. To obtain information about NHSMUG, send an e-mail asking for shareware distribution assistance to `nhsmug@aol.com`. Do not send any programs to this e-mail address; it is for information only. The NHSMUG will distribute your shareware game to the Internet and America Online if you ask for their assistance doing so.

Make sure to keep your address current in your games. You might want to consider getting a post office box if you plan to get serious about distributing shareware games. That way, if you move, mail can be forwarded easily to your new address. It sometimes takes a long time for a new version of your game to replace older versions that have old addresses in them. You can lose a number of registrations if the user sends the fee to an old address, only to have it returned as undeliverable.

Offering Incentives to Register Your Game

How do you get users to register your game? There are a great number of methods used by today's shareware games. Most methods require that you send something back to the user when they register. This is only fair because you are receiving payment. However, be prepared to handle the workload involved in responding to registrations.

Do not promise more than you can realistically handle. For example, it is generally unwise to promise sending out newer versions of your games to registered users for several reasons. First, you will probably be making a number of bug fixes and enhancements to the game as people report errors and send in suggestions. Second, if you start receiving a large number of registrations, perhaps in the hundreds, you'll need to send updates to a large number of people. That can get expensive quickly. Generally, it is best to promise the latest version of a game at the time the user registers. If you want to provide free updates, it is best to have the users contact you. That way, you will only have to send newer versions of your games to those who are truly interested in getting updates.

One of the more popular methods to get users to register is to distribute a demo, limited, or crippled version of your game. When the user registers, you promise to send them a code or new copy of the game, which enables all features of the shareware version. The trick here is to know how much to leave out of the game without making game play too limited. You want the user to get a chance to play the game for a while to determine how much they like it, while at the same time giving them something to look forward to when they pay the registration fee. For example, if you have written a role-playing or adventure game, maybe you can allow the user to play only the first two quests in the shareware version. For an arcade game with multiple levels, perhaps the shareware version of the game only allows you to play the first 10 levels. After registering, the entire game can be played.

Another common method used to encourage users to register is to include extended shareware notices in the game. These extended notices

may appear from time to time during game play, or display for several seconds when the game is first run or exited. The user must put up with these notices if they do not register the game. After registering, the user enters a code you provide to make the notices disappear. You must be careful when using this registration incentive because you do not want to annoy the user so much as to turn them off your game.

Some shareware authors who have written a number of shareware games offer to send registered users multiple games. In other words, if the user registers for one game, they will receive a disk containing more games written by the author. Of course, this method only works after you have written several games.

The National Home and School Macintosh Users Group has a reputation for offering shareware authors certain programs to get users to register shareware games. In the past, this has taken the form of discounts or free samples of products offered by the NHSMUG. When a user registers one of your games, you typically send a coupon to the registered user who can then send it to NHSMUG for their discount or free sample. You, in turn, could advertise such an incentive in your game, hopefully with the effect of obtaining more registrations. To see what the NHSMUG currently offers, send an e-mail message to `nhsmug@aol.com`.

Registration Fees

Typical registration fees for shareware games range from $10 to $25. Anything less than $10 probably isn't worth it for the user to send in the fee or for you to process it. Anything more than $25 and your shareware game starts becoming as expensive as commercial games. Because your games will be seen globally, make sure to denote the currency type, e.g., U.S. dollars. The average registration fee hovers somewhere around $15(U.S.). Your game should be something really special if you plan on asking for more.

Supporting Your Game

Writing and releasing a Macintosh game requires a certain amount of responsibility on your part. A good author will support a game after it is released. Support includes fixing bugs, responding to user's questions and comments, and enhancing the game in response to suggestions. The amount of time you spend supporting your game can have a great effect on how well that game succeeds in the market. Many games evolve over time, sometimes starting out little more than weekend hacks and ending up popular hits. Be prepared to spend as much time supporting your game as you did writing it.

If you just want to write a game and make it available to the world without having any additional responsibility, you can release your game as *freeware*. Freeware games are usually provided on an as-is and require no registration. You can still get feedback and suggestions, and build a reputation for yourself without the added responsibility of supporting your game. However, it is still good practice to fix any bugs that cause system errors. You wouldn't want to build a negative reputation for yourself.

Closing Comments

You now know just about everything that goes into writing a Macintosh game, from idea generation to coding to marketing and distribution. Try not to be overwhelmed by the skills needed to write a complete game. Take it one step at a time, learning things as you need to along the way. Take advantage of other programmers' experience by examining their source code and asking questions (see Appendix A for where to obtain additional information and help in writing games). Always keep in mind that writing games can be a lot of fun and often brings many rewards.

APPENDIX A

OTHER SOURCES OF INFORMATION

It would be impossible to teach you everything there is to know about Macintosh programming in one short book. You might find yourself needing additional information on topics discussed in this book, or even on topics not covered in this book. Where can you obtain this information? Good question. This appendix will list several resources available to Macintosh developers, and give you a head start in obtaining additional programming help.

Inside Macintosh Series

The Inside Macintosh Series is a collection of books written by Apple Computer, and is considered the reference of choice by Macintosh developers. If you want the official word on Macintosh programming topics, this series provides just what you need. The series itself now spans over two dozen books, where each book focuses on a particular toolbox manager or programming topic. You can typically find these books at a large bookstore, or you can order them in printed or electronic form from Apple's Developer Tools Catalog (which will be discussed shortly). The following books comprise the series as of the publication date of this book (new titles are added as the Macintosh operating system changes).

- 3D Graphics Programming with QuickDraw 3D
- Advanced Color Imaging on the Mac OS
- Apple Guide Complete
- AppleScript Finder Guide
- AppleScript Scripting Additions Guide
- AppleScript Language Guide
- Inside Macintosh: Overview
- Inside Macintosh: Macintosh Toolbox Essentials
- Inside Macintosh: More Macintosh Toolbox
- Inside Macintosh: Imaging With QuickDraw
- Inside Macintosh: Text
- Inside Macintosh: Files
- Inside Macintosh: Memory
- Inside Macintosh: Processes
- Inside Macintosh: Operating System Utilities
- Inside Macintosh: Devices
- Inside Macintosh: Interapplication Communication

- Inside Macintosh: Networking
- Inside Macintosh: QuickTime
- Inside Macintosh: QuickTime Components
- Inside Macintosh: Sound
- Inside Macintosh: X-Ref
- Inside Macintosh: AOCE Application Interfaces
- Inside Macintosh: AOCE Service Access Modules
- Inside Macintosh: PowerPC System Software
- Inside Macintosh: PowerPC Numerics
- Inside Macintosh: QuickDraw GX Programmer's Overview
- Inside Macintosh: QuickDraw GX Objects
- Inside Macintosh: QuickDraw GX Graphics
- Inside Macintosh: QuickDraw GX Typography
- Inside Macintosh: QuickDraw GX Printing
- Inside Macintosh: QuickDraw GX Printing Extensions and Drivers
- Inside Macintosh: QuickDraw GX Environment and Utilities

Just one final note on the Inside Macintosh series: Buying the printed form of all the books would make the national debt look small, but you can pick up a CD-ROM containing electronic versions of most of the books for less than $100. See the section on the Developer Tools Catalog for where you can buy the CD-ROM version of the Inside Macintosh Series.

Apple Technical Notes

Over the years many technical notes have been released, written by real programmers covering a wide variety of topics. These tech notes are available from the Internet or on the Bookmark CD included with *develop*, The *Apple Technical Journal* (see the section on *develop* for more information). Collections of tech notes are usually grouped by topic, making it relatively easy to locate ones that may be of interest to you.

The Apple Developer Catalog

The *Apple Developer Catalog* contains probably the most comprehensive list of development products available for the Macintosh. Everything from development environments to books to training materials to programming tools can be found here. To be sure, you'll pay retail price for these products, but everything you could ever want can be found here. To obtain a copy of the catalog, use one of the following means.

Electronic Mail

apda@applelink.apple.com

Telephone

U.S.	1-800-282-2732
Canada	1-800-637-0029
International	(716) 871-6555
Fax	(716) 871-6511

Mail

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

Web Site

http://dev.info.apple.com

Metrowerks and Symantec Development Systems

Metrowerks and Symantec offer two of the most popular development environments for the Macintosh computer (CodeWarrior by Metrowerks, and Symantec C++ and Think C by Symantec). If you are looking to obtain a development environment and you are a student or teacher, you can typically obtain a substantial educational discount. Contact the company directly to see if they are currently offering any educational discounts or special offers.

Metrowerks

Metrowerks Corporation
3925 West Braker Lane, Suite 310
Austin, TX 78597
(512) 305-0400 (voice)
(512) 305-0440 (fax)
(800) 377-5416 (order only in the U.S.)
`http://www.metrowerks.com` (web site)

Symantec

Symantec Corporation Headquarters
10210 Torre Avenue
Cupertino, CA 95014
(503) 334-6054 (voice)
(503) 334-7400 (fax)
(800) 441-7234 (U.S. and Canada only)
`http://www.symantec.com` (web site)

develop, The Apple Technical Journal

This quarterly journal contains articles, columns, and question and answer sections covering a wide range of Macintosh programming topics. Included with each issue is the Bookmark CD, a CD-ROM containing a wealth of programming documentation, examples, source code, and related information. You can purchase individual copies for $10, or subscribe for a whole year for around $30 (in the U.S.). Considering the information contained on the CD-ROM alone (including several Inside Macintosh books, all the tech notes, and programming examples, including some for games), this is one of the best Macintosh programming values around. To subscribe, simply contact the *Apple Developer Catalog* via one of the previously listed means.

Usenet Newsgroups

There are several Macintosh games and programming-related newsgroups available on the Internet. Make sure to look over the FAQs (frequently asked questions) before posting to a newsgroup in order to make sure you understand the guidelines and ground rules about what's appropriate to discuss on each newsgroup. These newsgroups are frequented by programmers and game players alike, and provide a great source of information specific to the design and development of games and other programs for the Macintosh computer. You'd do yourself a great favor by reading over some of these newsgroups.

- `comp.sys.mac.programmer.help`
- `comp.sys.mac.programmer.info`
- `comp.sys.mac.programmer.misc`
- `comp.sys.mac.programmer.tools`
- `comp.sys.mac.programmer.games`
- `comp.sys.mac.games.action`

- `comp.sys.mac.games.adventure`
- `comp.sys.mac.games.announce`
- `comp.sys.mac.games.flight-sim`
- `comp.sys.mac.games.marketplace`
- `comp.sys.mac.games.misc`
- `comp.sys.mac.games.strategic`

Note that these newsgroups can sometime change names over time or disappear completely. Also note that new newsgroups can come into being. Make sure to take advantage of your newsreader's newsgroup search facility to get the latest up-to-date names of Macintosh games and programming-related newsgroups.

Web Sites

There are already several Macintosh programming-related World Wide Web sites available today, and many are being added as you read this book. Due to the volatile nature of the locations of these web sites, it would probably be silly of me to list any specific sites here because they may not be around or may have different names and locations by the time you read this book. However, you can search for these sites yourself. One of the best places to start is the Yahoo Web page (`http://www.yahoo.com`). Specify "Macintosh and Programming" as the search criteria, and you'll be rewarded with a list of the Web sites that cater to Macintosh programmers.

National User Groups

If you plan to distribute your game as shareware, you would do yourself a great service by sending a copy of your program to the following national users groups. They each boast a large membership and vast software collections.

Arizona Macintosh Users Group

Arizona Macintosh Users Group
Attention: Author Submissions
4131 North 27th Street, Suite A120
Phoenix, AZ 85016
America Online: AMUG
E-mail: `info@amug.org`
Web site: `http://www.amug.org`
Phone: (602) 553-0066

Berkeley Macintosh Users Group

Berkeley Macintosh Users Group
Attention: Art Lau
1442A Walnut Street, #62
Berkeley, CA 94709
E-mail: `art_lau@bmug.com`
Web site: `http://www.bmug.org`
Phone: (510) 549-2684

National Home and School Macintosh Users Group

National Home and School Macintosh Users Group
Attention: Shareware Distributions
P.O. Box 64064
Kenner, LA 70064
E-mail: `nhsmug@aol.com` (information only, do not send programs here)

APPENDIX B

ABOUT THE CD

The CD-ROM included with this book contains the complete Desert Trek source code, a demo version of the Symantec C++ development environment for the Power Macintosh, several programming tools, and shareware games for fun and study. Given the last-minute nature of putting together items to be included with the CD-ROM, make sure to look over the Readme file contained on the CD-ROM for any last-minute updates.

Desert Trek Source Code

The entire source code and all supporting files for Desert Trek can be found in the Desert Trek Source folder. This includes project files for Think C and CodeWarrior, graphics files used to draw Desert Trek's graphics, the resource file containing all sounds, dialog boxes, and other resource data, and all source code and header files. Because nearly all of the examples given in this book come from Desert Trek's source code, you may want to copy the project and source files to your Macintosh's hard drive. You will need to copy the source code folder, the resource file, and the project file corresponding to the development system you're using. If you plan to use the demonstration version of Symantec C++ version 8 included on the CD-ROM, the Think C version 7 project file for Desert Trek will be converted for you when you first open it.

The source code for Desert Trek has been saved in text format, meaning that you can look at it using any text editor. Feel free to borrow portions of the code for use in your own games.

Demonstration Version of Symantec C++ for the Power Macintosh

A demonstration version of Symantec C++ is included on the CD-ROM so that you can compile the Desert Trek project even if you don't currently own a development system. Note that you must have a Power Macintosh in order to use this version of Symantec C++. For installation instructions and additional information, see the documentation included on the CD-ROM.

Programming Tools

Several shareware tools have been included on the CD-ROM that you might find useful in the development of games. If you plan to use these tools, please compensate the authors of these fine programs for their

efforts by registering them. The following are descriptions of two of the tools I used to create Desert Trek.

SoundMacer by Ingemar Rangnemalm

SoundMacer is an excellent and easy-to-use digitized sound utility that can dramatically reduce the size of your 'snd ' resources without much sacrifice in sound quality.

3D Buttons CDEF by Zig Zichterman

The 3D buttons used in Desert Trek are compliments of the 3D Buttons CDEF. See the Readme file and Chapter 3 on Resources to see how you can use the 3D Buttons CDEF in your games.

Shareware Games

Several shareware games are included on the CD-ROM so that you can get an idea of the types and styles of shareware games available for the Macintosh (well, okay, for a little fun too). Remember to compensate the shareware authors for their efforts by registering, if you plan to make the game a part of your software collection. Keep in mind that you'll be on that side of the fence if you release your games as shareware.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

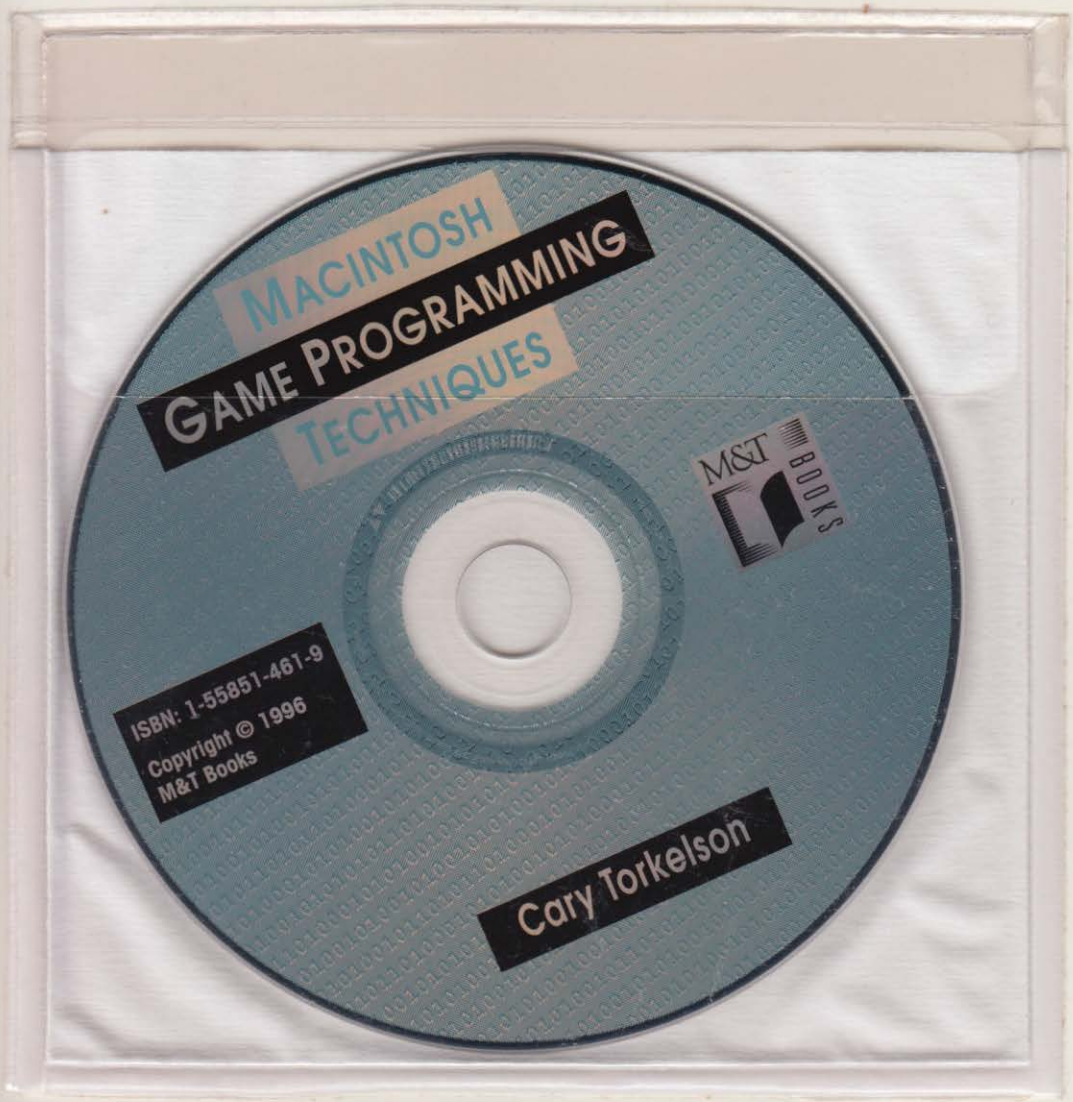
U

V

W

Y

Z


MACINTOSH
GAME PROGRAMMING
TECHNIQUES
M&T BOOKS
ISBN: 1-55851-461-9
Copyright © 1996
M&T Books
Cary Torkelson

Macintosh Game PROGRAMMING Techniques

Macintosh Game Programming Techniques is an excellent introduction to the world of programming games for the Macintosh. Programming games requires a mindset and a set of skills different from those required by almost any other type of application development. Game programmers need to focus on graphics, sound, and interacting with the user in real-time. The Macintosh also adds additional tools for the programmer to learn—like QuickDraw, A5 pointers to global variables, and memory management techniques different from other platforms.

Inside, you'll learn how to re-create the "look and feel" that Mac users have come to expect. You'll also create a complete game from start to finish incorporating everything a game should have—graphics, sound, and game intelligence. You'll discover how to let the game interact with the user, without freezing due to excessive input. You'll learn the techniques that are required to program for the Macintosh like using Toolbox routines, QuickDraw, and digitized sound.

CARY TORKELSON is a design analyst for Dean Witter and a prolific Macintosh game Shareware author. He is a regular columnist with *Home and School Mac*, a magazine published by the National Home and School Macintosh User Group.

- **Learn to program game intelligence**
- **Create user interaction without overloading input**
- **Incorporate digital sound with your program**
- **Master dialogs, modeless dialogs and other forms of user input**
- **Create menus that your users will find natural and easy to follow**
- **Use 32-bit QuickDraw including CopyBits and CopyMask to create exciting graphics**
- **Manage memory efficiently**
- **Create and manage resources effectively**

CD-ROM INCLUDED

The CD-ROM contains the complete source code for Torkelson's popular game DesertTrek as well as Symantec C++ Lite for the Power Macintosh. Also included are additional games demonstrating different game design techniques including Galactic Empire, Pegged, and MacMines.

Level

Beginner/Intermediate

Programming

Macintosh

Cover design by Gary Szczecina
Cover art by Peter Kuper, The Image Bank

M&T BOOKS

ISBN 1-55851-461-9

US $39.95
CAN $54.00
90000>

9 781558 514614