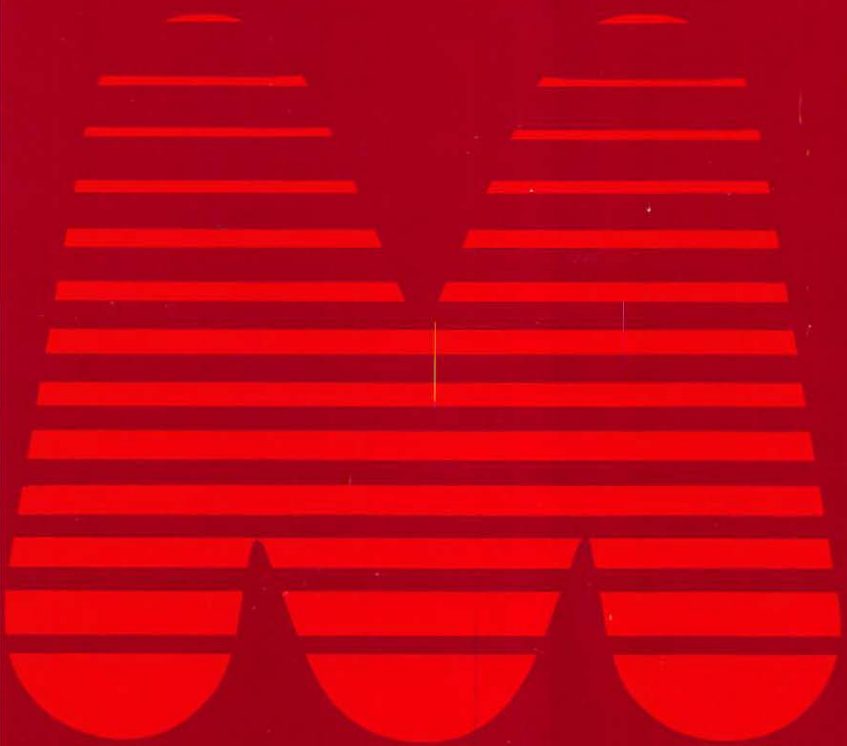
The Waite Group

HIDDEN POWERS OF THE MACINTOSH®

PLUME
WAITE



The complete guide to Macintosh® Software Development

- A simplified approach to how the Mac *really* works
- Demystifies the Quickdraw and Toolbox ROM routines
- Explains GrafPorts, bit-maps, regions, clipping, shapes
- Develops actual applications programs
- Covers window and menu management, events, and resources

by Christopher L. Morgan

Another book by the bestselling authors of ASSEMBLY LANGUAGE PRIMER FOR THE IBM® PC & XT and BLUEBOOK OF ASSEMBLY ROUTINES FOR THE IBM® PC & XT .... And rave reviews for The Waite Group:

"Anyone who is new to assembly language programming on the IBM PC and feels completely at sea will find a welcome port in Robert Lafore's Assembly Language Primer for the IBM PC & XT."

—John Figueras, reviewing
Assembly Language Primer for the IBM PC & XT, in *Byte*

"[Chris Morgan] has succeeded in producing a volume that no assembly language programmer can do without."

—*Bluebook of Assembly Language Subroutines*,
reviewed in *The Reader's Guide to Microcomputer Books*

"An outstanding example of how to write a technical book for the beginner... refreshingly enjoyable ... accurate, readable, understandable, and indispensible. Don't stay home without it."

—Ken Barber, reviewing
CP/M Primer, in *Microcomputing*

"Mitch Waite ... has left a distinctive contribution to the literature of computer graphics ... seeing it here is like understanding it for the first time."

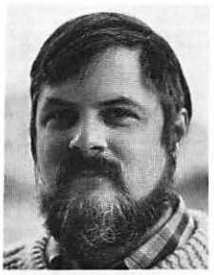
—*Computer Graphics Primer*, reviewed in
Computer Graphics World

"It's hard to imagine that a field only a decade old already has a classic, but Waite's book is just that."

—Tony Dirksen, reviewing
Computer Graphics Primer in *Interface Age*

"... does an excellent job of demystifying the whole study of computer programming in BASIC."

—Annie Fox, reviewing
BASIC Primer in *Creative Computing*



Christopher L. Morgan is a professor at California State University, Hayward, where he teaches mathematics and computer science, including computer graphics, assembly language programming, computer architecture, and operating systems. Dr. Morgan has given talks and authored papers in pure mathematics and on representations of higher-dimensional objects on computers. He is director of the computer graphics lab at Hayward and is a member of a number of professional associations, including the American Mathematical Society, the National Council of Teachers of Mathematics, and the Association for Computing Machinery. He is coauthor along with Mitchell Waite of *8086/8088 16-Bit Microprocessor Primer*, *Graphics Primer for the IBM® PC and XT* and *Hidden Powers of the TRS-80® Model 100*.

Christopher L. Morgan

Hidden Powers of the Macintosh®

A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

PLUME TRADEMARK REG. U.S. PAT. OFF. AND FOREIGN COUNTRIES
REGISTERED TRADEMARK — MARCA REGISTRADA
HECHO EN WESTFORD, MASS., U.S.A.

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L1M8

Library of Congress Cataloging-in-Publication Data

Morgan, Christopher L.
Hidden powers of the Macintosh.

"A Plume/Waite book."
On t.p. the registered trademark symbol 'R' is superscript following "Macintosh" in the title.
1. Macintosh (Computer)—Programming. I. Title.
QA76.8.M3M67 1985 005.2'65 85-15514
ISBN 0-452-25643-7

Interior design by Rick Chafian

Typography by Walker Graphics

First Printing, October 1985

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

To my family

Contents

Acknowledgments

There are a number of people to whom I am indebted for their help in making this book possible.

The Waite Group has provided invaluable support. Mitchell Waite initiated and oversaw the entire project. Robert Lafore has served admirably as technical and managing editor. Kim House provided technical support. Lyn Cordell oversaw the production, and Joan Frank handled the details of production.

Steve Jasic provided a copy of his disassembler to give me another view of the Macintosh's ROM.

The people at Apple provided development support including preliminary copies of software and technical reference manuals.

My wife Carol and my children Elizabeth and Thomas patiently supported me while I worked on this book.

Preface

The Macintosh is a revolutionary computer. It is the first reasonably-priced computer that can be called "human-oriented" — that is, easy for humans to use, rather than easy for engineers to build. However, this ease of use has been achieved at a price: programming on the Macintosh — at least the writing of serious applications programs — is more difficult to learn than on previous microcomputers. This is because the Macintosh contains a truly fantastic collection of software routines, mostly built into its ROM, to control menus, windows, graphics, the mouse, and other parts of the Mac's operations. To write serious programs on the Mac, you must learn about these routines, and about the concepts they are based on. In short, you must learn about the Mac's "Hidden Powers."

This book explains how the Macintosh's built-in software works and how to write applications programs that work with and take advantage of these built-in routines. The aim is to teach the philosophy and organization of the Macintosh and its various system parts in an orderly, step-by-step approach, so that understanding and learning to program the Mac becomes as simple as possible.

Who This Book Is For

This book is intended for anyone who wants to learn how to write serious applications programs for the Macintosh as well as those simply interested in how the Mac works.

Although the book is introductory in nature, it is not intended for the computer novice. The Pascal language is used to demonstrate the various concepts, so you should have some exposure to Pascal or a modern, structured computer language like it. However, you are not expected to be

a Pascal expert. In fact, we explain those features, such as pointers and data typing, that make Pascal different from other languages, yet are essential to understanding the Macintosh.

You should have some familiarity with computer operating systems and computer operations in general, including such features as files, records, fields, interrupts, RAM, and ROM. Understanding the hexadecimal numbering system and hexadecimal memory addressing is also helpful. It's useful, too, to have some understanding of what assembly language does, although you by no means need to be an assembly language programmer to follow this book.

Finally, you should be familiar with the Macintosh's revolutionary features such as windows, menus, dialog boxes, and the mouse, and how they work together to create this amazingly easy-to-use machine.

Although the example programs in this book were developed using the Lisa Pascal Workshop (the original development system for the Mac), they are written and explained so that they are useful no matter what program development system you are using. Thus, whether you program in C, Pascal, assembler, Forth, or some other language, you will still find that this book reveals the fundamental concepts of the Mac's operation in a way applicable to your particular programming environment.

How This Book Is Organized

This book is organized around example programs. This has the advantage of providing a concrete basis from which to launch the more abstract concept descriptions. Each program is designed to show how several built-in Macintosh routines integrate into a working unit. By seeing how these routines work together, you will come to understand the ideas behind them and the overall philosophy behind the Macintosh design. In the case of the Macintosh, it is really true that the whole is greater than the sum of the parts.

The first three chapters lay the groundwork, discussing the organization of the Mac in general terms. Subsequent chapters provide a series of programs that demonstrate the ideas behind the Mac's operation.

Chapter 1 explains the basic philosophy and history behind the Macintosh. It explains how Macintosh's windows and menus originated in research at Xerox's Palo Alto Research Center.

Chapter 2 surveys the Macintosh's internal organization, explaining how the hardware "maps" to the memory addressing space and how the RAM and ROM are allotted to the various parts of the Operating System. This chapter also surveys the logical division of the Operating System into *managers*.

Chapter 3 discusses several fundamental aspects of the applications programming environment for the Macintosh, including how applications are constructed as collections of resources, some "pointers" on Pascal, and a debugging session.

Chapter 4 introduces the details of *QuickDraw*. A series of short example programs introduces the basic features of QuickDraw — from a simple program that does no more than initialize the drawing environment to a program that shows how to control a number of pictures on the screen at once, a precursor to multiple windows.

Chapter 5 introduces the *Event Manager*. An example program illustrates how the mouse and the keyboard can generate *events*. You see how the Event Manager handles these events, presenting them to the applications program as it is ready to handle them. You see how the facilities of QuickDraw track the mouse as it enters certain *regions* on the screen. This is a precursor to *controls*.

Chapter 6 introduces the *Window Manager* and the *Control Manager*. An example program illustrates how to track the mouse through parts of a window, such as its *drag region*, *goAway box*, and *grow box*. It also illustrates how controls such as *scroll bars* work.

Chapter 7 extends the concepts introduced in chapter 6 to multiple overlapping windows. It introduces QuickDraw features such as *polygons* and *pictures*.

Chapter 8 extends window management one more step by introducing the *Dialog Manager*. The example program illustrates how to use the high-level management routines in this manager. We show you how to call a single routine to handle entire interactions between the Macintosh and the user. We also show how to share control of such interactions between the program and the system.

Chapter 9 introduces the *Menu Manager*. An example program illustrates how to create menus and how to track the menu selection process. This example also illustrates some basic shapes and drawing attributes available in QuickDraw.

Chapter 10 introduces the *File Manager* and *Text Edit*, the manager of text. An example program assembles these techniques, illustrating how a text editor applications program can be built that loads and saves text files from and to the disk. The program also uses the managers introduced in previous chapters, including QuickDraw, the Event Manager, the Window Manager, the Control Manager, and the Dialog Manager. This final example illustrates how these different system parts work cooperatively.

With one exception, the example programs use features from current or previous chapters. The example programs illustrate the core features of Macintosh's built-in software, leading up to the final example in Chapter

10, which in many ways is a typical applications program. The purpose of these examples is to provide the simplest and clearest demonstration of the Mac's features, providing models for developing your own software. We are not attempting to create a cookbook of stand-alone utilities.

All examples have been carefully coded to work closely with the Macintosh's built-in software. Each line has been checked for its role in making the example work right.

Each example is fully explained in the text. These explanations introduce the basic concepts of each chapter. Because the explanations are closely linked to the examples, you see in concrete terms what these concepts mean to an applications programmer.

The Macintosh contains a wealth of software involving an immense number of concepts. It is thus impossible to cover every detail of the Mac's operation in a single book. In view of this, we have taken the approach of carefully selecting key concepts, guiding you to an overall, solid understanding of how the Macintosh works. From this foundation, you should be able to deal easily with its other, less critical features.

We hope you find this book a unique and useful approach to the Macintosh, one that leads you to understanding how the Macintosh works and how to write programs that run on the Mac, and taking full advantage of its features.

1

The Origins of the Macintosh Design

This chapter covers the following new concepts:

- **The Human Interface**
- **Smalltalk: Menus, Windows, and the Mouse**
- **Object-based Systems**
- **Modes**
- **Editing**
- **The Lisa**
- **QuickDraw and the Macintosh Managers**

On January 24, 1984, Apple announced the Macintosh computer to its shareholders and the world, ushering in the most revolutionary and, for the programmer, one of the most complex machines of its time. In this chapter, we discuss the philosophy that underlies the Macintosh's design. We trace its origins to the early 1970s, when researchers first envisioned the windows and menus that distinguish the Macintosh's user-friendly, graphics-oriented operating system. We explain why the Macintosh uses windows and menus, why its Operating System is divided into managers, why so much software is built into the Macintosh, and what this means to the user and programmer.

The Idea of a Human Interface

The intent of the Macintosh is to provide a natural and powerful setting for users to work in. It is, as Apple puts it, "the computer you already know how to use". Probably, it will be used mostly to produce documents such as reports, notes, manuscripts, memos, and letters. The Macintosh can also compute for such things as taxes, household expenses, and mathematical or scientific problems.

The Macintosh can generate graphics either through a "paint" program or by a program written in a high-level language such as BASIC or Pascal. These graphics can then be fully integrated into documents so that text and diagram appear on the screen exactly as they will be printed. Text can be displayed and printed in many sizes and styles, including a variety of fonts and effects, such as bold or italic.

Much of the Macintosh's capabilities revolve around its approach to what is called the "human interface". It uses a crisp, high-resolution screen and a mouse to present the user with a highly interactive work environment.

The computer can display images that put the user in familiar settings. For example, the initial screen appears as a desktop with little images of objects, such as file folders and sheets of paper (see Figure 1-1). These icons (small pictures that represent real objects) show the possible actions, and the mouse allows the user to quickly select a particular option. Complicated syntax for commands need not be looked up or memorized. This greatly increases productivity.

The Origins of the Macintosh

The Macintosh represents the first low-cost implementation of ideas developed during the 1970s at Xerox's Palo Alto Research Center (PARC). It was there that Alan Kay and others started to experiment with personal computers before such machines could realistically be built.

Although it was not possible in the early seventies to build a computer that was both affordable and sufficiently powerful, the researchers at Xerox PARC decided to build prototype machines in any way possible. They had faith that hardware costs would continue to decline. They knew that improving chip technology would make the ideas, which could then be implemented only at ridiculously high cost, eventually quite practical for a desktop or home computer.

The researchers at Xerox brought bright students from nearby schools into their labs to test new ideas as fast as the appropriate hardware and software could be built. The researchers also used these machines to

accomplish their own work, such as designing and developing the software and writing articles. The goal was to build machines that worked well for everyone, not just the technically oriented.

We briefly explore here some of the ideas developed at Xerox PARC, since they provide a background to many of the concepts in the Macintosh and may aid in understanding some reasons behind the Macintosh design.

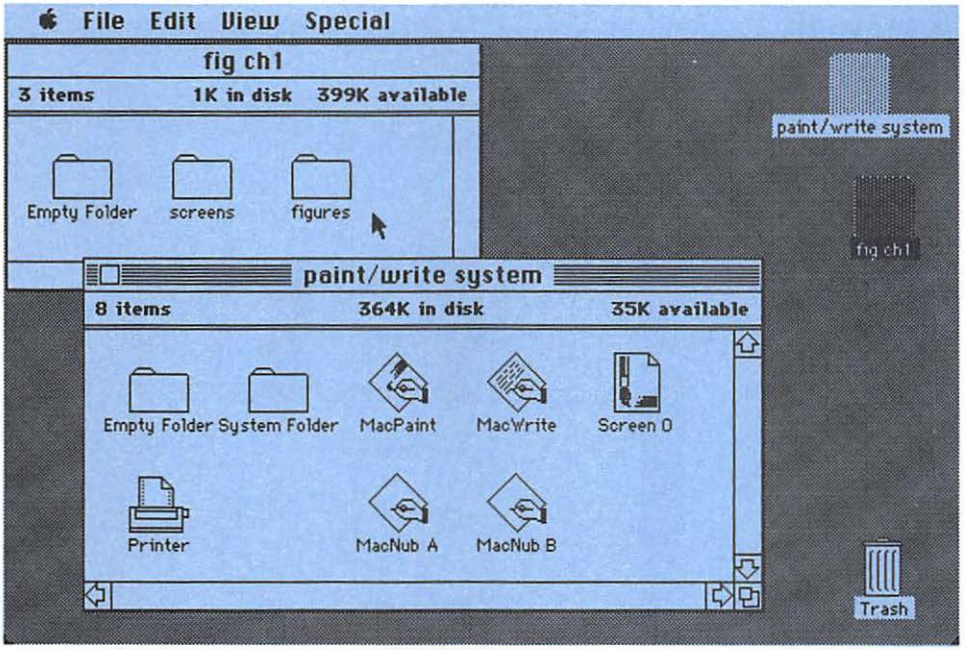
Smalltalk

The language Smalltalk was developed by these researchers. Smalltalk has a number of interesting features, including a high-resolution screen, a mouse as an input device, multiple overlapping windows, pop-out menus, and an object-based programming structure. It also incorporates some basic approaches to text editing. Let's look at these features in more detail.

The Screen and the Mouse

The display screen for Smalltalk is "bit mapped": that is, each dot on the screen is individually controlled by a bit in memory. Thus, the programmer has control over every dot on the screen. This type of display is desirable

Figure 1-1. A Macintosh Desktop



because it allows both text and graphics to be drawn and "integrated". The Macintosh also uses a bit-mapped screen. Later, we explore how this screen works.

A Smalltalk mouse generally has three buttons. One button acts as a *selector* for windows and their contents. The other two buttons are for menu activation and selection. In Smalltalk, menus are often "hidden" when they are inactive; they need to be made active for you to see them.

In contrast, the Macintosh's mouse has only one button for all three purposes. This is possible because Macintosh menus are always accessible through the menu bar along the top of the screen.

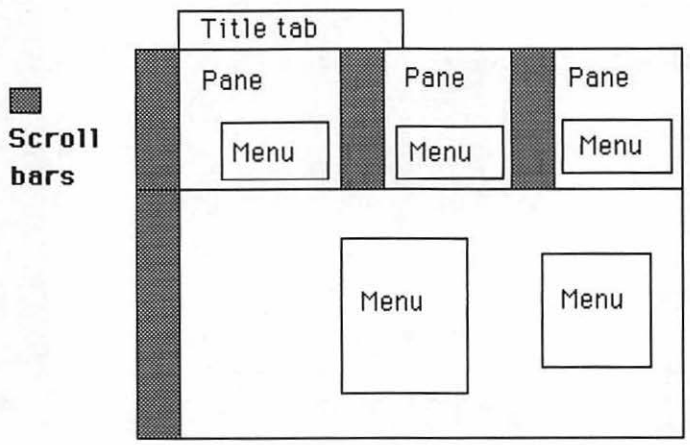
The mouse was invented by Douglas Engelbart in 1964 to help people better input "xy-position" information. Subsequent research proved it to be superior to other systems, such as cursor keys, digitizing tablets, and touch screens.

Smalltalk Windows

A Smalltalk *window* is a rectangular area with a *title tab* sticking up from the top edge (see Figure 1-2). Each window can be divided into a number of *panes*, each with its own *scroll bar*. Each window has a *window menu* that pops up, usually in response to pressing one of the mouse buttons. Each pane has its own pop-out menu. When a menu is not "popped up", it is normally invisible.

Each Smalltalk window displays information for a particular *task* or program. The user can have many windows on the screen at once and

Figure 1-2. A Smalltalk Window



thus is able to see the condition of many tasks at any moment. Having many windows open at once allows the user to work in a natural way, examining and working with many things at once. The fact that the windows overlap is crucial to efficient operation, since older windows need not be moved or resized as new windows appear.

The Macintosh's windows are not related to separate tasks, but instead display different information about the same task. That is, where Smalltalk uses several panes within a window, the Macintosh uses several windows on the screen. However, the Macintosh does have *desk accessories*, which are separate system *tasks* that can be run at the same time that an applications program is running.

In Smalltalk, only one window (and thus task) is active at a time. The user can activate a window by moving the cursor to that window and clicking the *selector* button. When a window is selected, it comes to the front of the screen and is seen in its entirety. Only one pane of a window is active at any one time. The user operates the mouse selector button to activate panes.

Window panes can work cooperatively, allowing the user to make a series of choices along a decision tree. The choice in the first pane may affect the choices in the second pane, and so on.

Smalltalk may be programmed by making menu selections and by typing text into window panes. The programming process is like filling out forms. The machine helps the programmer by either providing the proper menu or displaying explanations of what is required.

Objects

Smalltalk is object-based. That is, instead of separate data structures and procedures, it has *objects*, which package sets of data structures containing procedures that work on that data. This offers a special layer of protection, since data cannot be corrupted arbitrarily by a part of the system not "trained" to deal with that data. This also makes it easy to run at once a number of "jobs" that share information.

A Smalltalk program consists of *messages* that are sent to objects. Each object has a specific set of messages that it understands. The messages specify *what* is to be done but don't specify exactly how. If a message is received that is not appropriate, an error message is issued saying that the original message was not understood.

The Macintosh is definitely not object-based. Instead, it is designed to perform one thing at a time with much less protection of data. The exception is its desk accessories, which run independently but must be

explicitly given little slices of time by the applications program. However, as we shall see, the way the Macintosh is divided into *managers* traces its origins to Smalltalk's object-based approach.

Modes

A basic motivation for developing Smalltalk's multiple overlapping windows was to eliminate "modes". A mode is a set of options available to the computer user. For example, while in the editing mode, the user can perform a variety of editing commands.

Traditionally, a computer presents the user with a series of modes in which to work. Only one mode is active at a time. For example, the user may start in the Operating System command mode, move to the text editing mode, then enter the file transfer mode. Within these modes are other modes. For example, within the text editing mode are command modes, insert modes, and search modes.

Larry Tesler, a researcher at Xerox PARC and later one of the developers of the Macintosh, has observed that one of the most frustrating and counterproductive features of computing is modes. In "The Smalltalk Environment"(*Byte Magazine*, August 1981), he explains that programmers and secretaries alike complain about modes. Two chief problems occur: 1) while in one mode, the user cannot do something that is available in another mode, and 2) a particular action has different effects depending on which mode the system is in.

Smalltalk's overlapping windows solve this problem to some extent. Each window is, in some sense, in a mode. However, the user is given visual clues to which window is active and which windows are immediately available. The ability to rapidly move from one window to another and then back to the original window reduces the "unavailability" problem. And a window's menus and text make its set of options easy to inspect and select, reducing the "multiple effects" problem.

"Modeless" Text Editing

Text editing presents some very interesting problems relating to modes. In particular, the process of searching and replacing strings of text can put the user in modes where command keys act differently than usual.

The developers of Smalltalk invented a "modeless" method of text editing, using the ideas of "cut", "copy", and "paste", along with the idea of selection range. The selection range is always clearly indicated on the screen because it is highlighted. The cut command removes the text in the selection range, putting it into a special edit buffer. The paste command moves text from this buffer, replacing what was in the selection

range on the screen with the text in the edit buffer. The copy command copies the current selection range into the edit buffer without removing it from the screen.

Apple

In the late seventies and early eighties, Apple began research that led to the Macintosh family of computers. Originally, two separate teams at Apple worked on two different computers in this family: the Lisa and the Macintosh. Many involved were knowledgeable about Smalltalk and the work at Xerox PARC. At one point, Smalltalk was even implemented on the Lisa, but it was too slow to be practical.

The Lisa

The people at Apple began work on the Lisa during the early eighties. Bill Atkinson was the main designer. The Lisa has a high-resolution display screen, a single-button mouse, an MC68000 processor, and a hard disk. It included an extensive set of drawing routines called "QuickDraw", developed by Atkinson.

The Lisa was designed mainly as an office tool for document preparation, but it also has facilities for program development. The Lisa Office System environment presents the user with a "desktop" with icons representing the hardware and software that is available. Some of these icons are a wastebasket, a clock, a hard disk, a floppy disk, a pad of "stationery", a drawing program, and a text editing program. The mouse is used to select what is wanted. For example, to view and set the clock, click the mouse on the clock icon and the clock "opens" on the screen.

The Lisa Pascal Workshop provides program development facilities. Here, the programmer is presented with a less graphic "user interface". Series of text menus appear across the top of the screen. The user/programmer selects menu items by hitting keys on the keyboard. However, the edit option "opens" a program that displays multiple overlapping windows in which text can be cut, copied, and pasted.

The Lisa Pascal Workshop can be used to develop applications programs for both the Lisa and the Macintosh. The exact method of transporting programs from one machine to another has evolved from using a serial communications line to having compatible 3½ inch Sony disk drives.

In early 1985, the Lisa was merged with the Macintosh and called the Macintosh XL.

The Macintosh

About 1979 Apple began work on an "appliance" computer that was to become the original Macintosh. At first, it was an eight-bit machine with not enough power. In 1981, Steve Jobs, one of the founders and chairman of the board of Apple, arranged for the Lisa efforts to be brought over to the Macintosh project. Burrell Smith was able to put together the hardware with the MC68000 processor configuration the way it is today.

QuickDraw. Bill Atkinson moved from the Lisa project to work on the MacPaint applications program for the Macintosh. He moved QuickDraw from the Lisa to the Macintosh, greatly reducing its size while maintaining compatibility between Lisa QuickDraw and Macintosh QuickDraw.

QuickDraw was originally written by Atkinson in Pascal, consuming about 160K bytes of memory. A reduction to about 24K bytes was accomplished by transforming it into assembly language. QuickDraw is a collection of over 100 drawing commands. It includes routines to draw lines, rectangles, ovals, polygons, and irregular shapes, outlining or filling them with a variety of pen sizes and patterns.

The Managers. The Macintosh software was developed as a collection of software modules called *managers*. Each manager is in charge of a specific area of responsibility within the machine. For example, file input and output are handled by the "File Manager", and you can think of QuickDraw as the Screen Manager.

An applications program calls upon these different managers to get things done. In some sense, the managers are like the objects of Smalltalk, each with its own data and procedures. However, the formal construct of Smalltalk *messages* is missing, and data belonging to one manager can be accessed by other managers and applications programs (thus is not fully protected). But much of the data is protected to a considerable extent. For example, if you wish to move a window or change its size, you call the Window Manager to do this for you.

The managers range from the fairly standard File Manager to a Dialog Manager, which acts as a mini-applications program, handling all interactions between the user and the machine for an extended period.

A crucial decision was to freeze the Operating System software in ROM, including QuickDraw and all managers. This ensures that all applications act basically alike, reducing the problem of "modes". That is, by placing the basic menu selection, window drawing, and text editing routines in ROM, all applications programs respond to the user in the same manner. Thus, the user needs only one set of basic commands for

all applications, reducing the "multiple effects" problem (same user action causing a different effect depending on the mode).

The fact that so many low- and medium-level routines have already been written and built into the Macintosh relieves the applications program of responsibility for developing lots of code. At the same time, it forces the programmer to understand more than he or she ever wanted to know about a rather complex set of rules and ideas. This book should make these ideas a lot clearer.

Desk Accessories. Another feature of the Macintosh is its desk accessories. These are special built-in applications programs that can be run from each application, providing a wide set of facilities to users no matter where they are in the system. This was designed to reduce the unavailability problem associated with modes.

The Macintosh was not set up to run several applications programs at once with separate windows, as in Smalltalk. However, the desk accessories provides a small taste of what that is like.

Outside Development. Apple also encouraged outside developers. It put together a development kit that included a set of programs and files to run on the Lisa with volumes of documentation.

One emphasis in this documentation is the need for each application to present the user with a uniform way of doing things so that the entire Macintosh system remains well integrated and thereby easy to use. The example programs in this book adhere to these standards except as noted in our explanations about them.

Summary

In this chapter, we have introduced the Macintosh, explaining the basic ideas behind its design, including the mouse, high-resolution video screen, icons, windows, and menus, all of which are designed to help the user work efficiently.

We have explained how the Macintosh's designers have tried to eliminate the problem of modes by providing a uniform human interface and by providing desk accessories. We have also discussed the way the software is built in and how it is organized into managers for ease of design and programming.

2

Macintosh System Organization

This chapter covers the following new concepts:

- Physical and Logical Structure
- RAM and ROM
- System Static Variables
- Exception Vectors
- System Communications Area
- System Dispatch Table
- Manager Globals
- The Heap
- System and Application Zones
- Stack Area
- Video and Sound Areas
- Hardware Connections
- Unimplemented Instruction Codes

This chapter presents an overview of the Macintosh's built-in software and its internal organization. The chapter describes the reasons behind Macintosh's organization, what the various components are and how they

fit together. We start by describing the physical components and structure of the Macintosh and work toward a description of its logical structure. (By "physical" we mean such things as the layout of memory address space.) This approach is designed to keep you firmly anchored in spite of some rather abstract ideas behind the Macintosh's organization and operation.

In our description of the Macintosh's physical structure, we start with the locations of the RAM, ROM, and connection to hardware devices. We see how the RAM is divided into different areas, including an area for *static system variables*, an area for *dynamic variables*, a *stack*, and areas devoted to the video screen and the sound system.

We will see how the Macintosh's built-in software is accessed using the MC 68000 microprocessor's unimplemented instruction codes to, in effect, extend the processor's original instruction set to include about 494 new Macintosh instructions.

Finally, we survey the *logical* structure of the Macintosh's built-in software. We see how the Macintosh's built-in software is divided into *managers* and *drivers* that perform essential logical functions.

Memory Layout

This book has been written for 128K and 512K Macintoshes using version 1.1 of the Finder. (To find out which version of the Finder you have, select the "About the Finder" option of the Apple menu from the desktop.)

We discuss actual addresses for the 128K and 512K Macintoshes. However, we also discuss the "relative" placement of these addresses, as well as where the system stores "official" copies of the addresses. This way you and your program can easily find things, even on Macintoshes of other memory sizes and Finder vintages.

As we proceed, you will gradually understand the meaning of what is stored at these particular memory addresses. However, we often introduce them without a complete explanation because they are "landmarks" or point to "landmarks". Don't despair, their exact function is explained later.

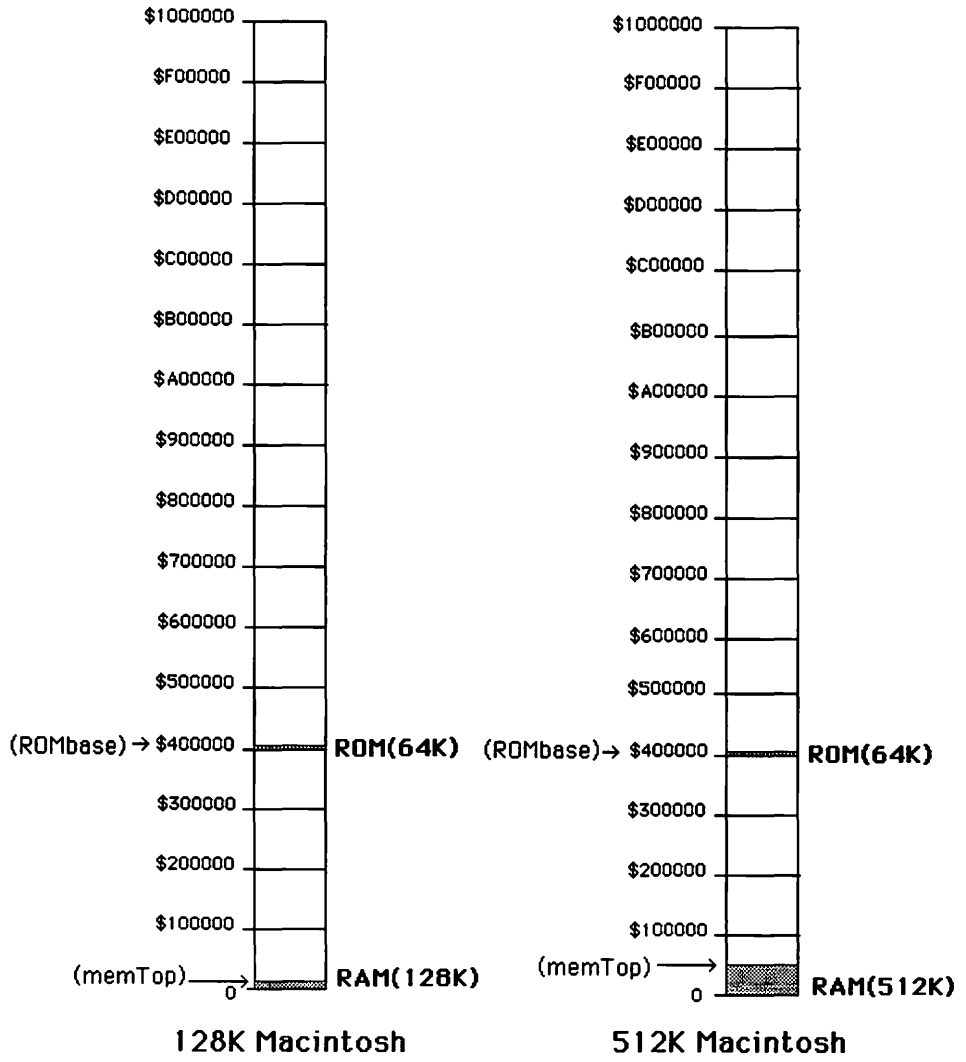
Physical Memory Layout

Physically, the Macintosh's 24-bit addressing capability gives it 16 megabytes of addressing space (see Figure 2-1). Memory stretches from $0 to $FFFFFF. (Note: The dollar sign indicates hexadecimal system notation in this book.) However, only a relatively small part of the Macintosh's total addressing space has RAM installed. This of course depends upon the

size of the Macintosh. On a 128K Macintosh, the RAM stretches from $0 to $1FFFF, and on the 512K Macintosh, to $7FFFF.

For both sizes of Macintosh, the ROM begins at $400000 and stretches 64K bytes to $40FFFF. This provides four megabytes beneath, leaving plenty of room for orderly expansion of RAM. On many machines, ROM occupies the lowest addresses. However, on the Macintosh, having RAM

Figure 2-1. The Addressing Space: RAM and ROM



there allows certain hardware-dependent locations to be manipulated, namely the *exception vectors*. This is useful when debuggers are used or when new "drivers" are installed.

Connections to the hardware, such as the disk controller and serial communications interface, use scattered locations in the upper half of the 16-megabyte addressing space from $800000 to $FFFFFF.

Now let's examine these various areas of the addressing space in detail.

The RAM

Let's start with the RAM (see Figure 2-2). At the lowest addresses is an area containing *static* system constants, variables, and tables. These quantities are called *static* because they do not move during the operation of the machine. Above this area is another area called the *heap*, which is managed *dynamically*. Data structures that grow and shrink in size can be placed here. For example, when windows overlap, cutting each other off in complicated ways, the data structure describing their visible part grows in size.

Above the heap is the *stack*, which grows downward toward the heap. Unlike the heap, the stack can only grow and shrink its data at one end. The Macintosh runs out of memory when the *stack* meets the *heap*. Above the stack is an area reserved for debuggers, and above that are areas for *sound* and *video*.

These three types of storage (static, heap, and stack) allow the Macintosh to take advantage of the best features of each, allowing it to place its data and code in just the right type of place.

System Static Variables Area

The lowest area of RAM contains the *System Static Variables Area*, which contains constants, variables, and tables needed by the Operating System and the Toolbox. As mentioned previously, these data structures are called static because they don't move around during machine operation. Many of these locations are referenced directly by the ROM; thus, they must be quite permanently fixed in position.

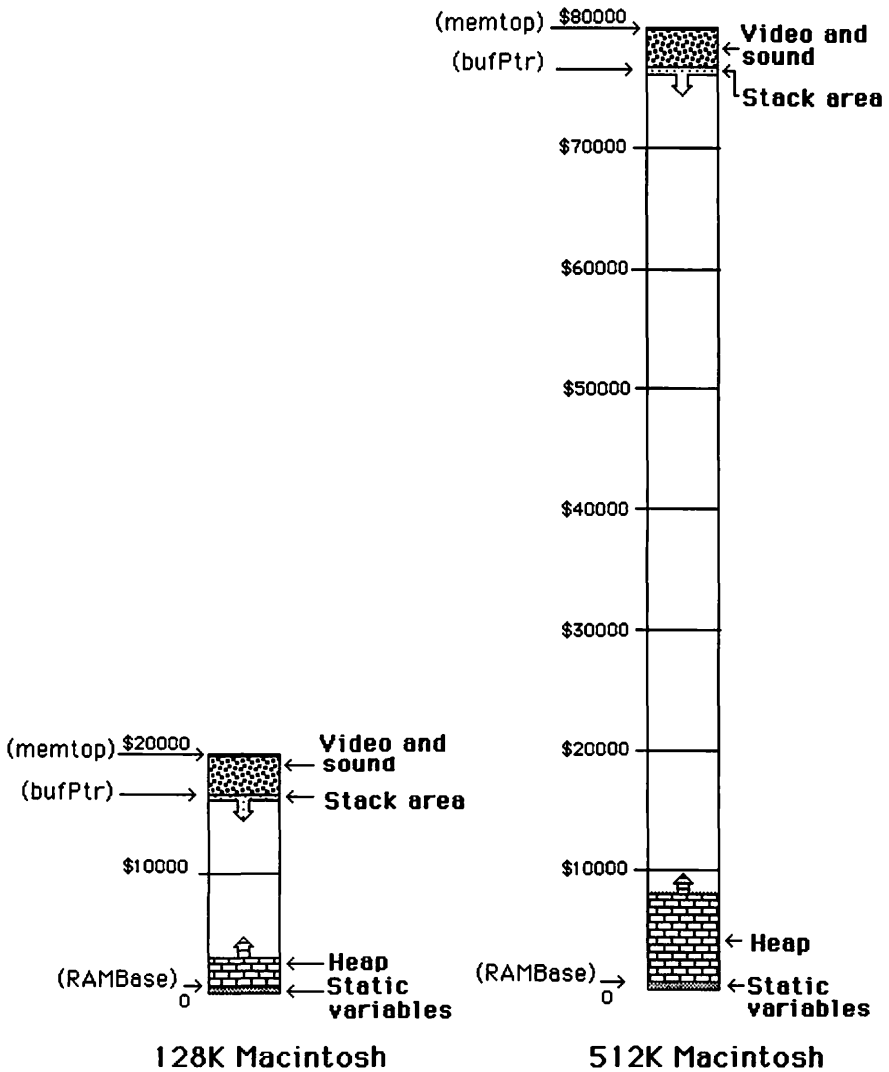
This area is further subdivided into sections, which we describe next (see Figure 2-3).

Exception Vectors. At the very lowest locations of the System Static Variables Area are the *exception vectors*. These extend from $0 to $FF and are an essential part of the operation of the 68000 processor. There,

locations are completely determined by the 68000 itself and are described in Motorola's documents on the MC68000 processor.

Each *exception vector* contains an address of a routine to handle a particular kind of *exception* to normal program execution. Some exceptions are generated by hardware *interrupts*, such as from the mouse or serial communications lines; some are generated by error conditions, such

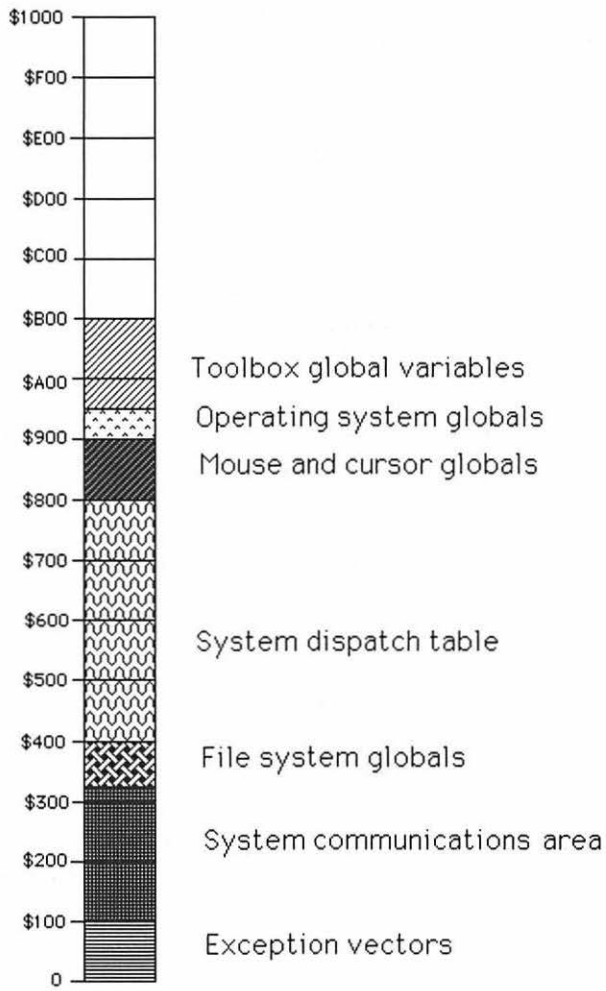
Figure 2-2. Layout of the RAM



as division by zero; and some are generated by unimplemented instructions used to access the Macintosh's built-in software. (We say more about this later.)

For now, understand that the position of the mouse is constantly and automatically updated through hardware *interrupts*. That is, whenever the mouse is moved, it generates interrupts that cause the Macintosh to compute its new position. Other interrupts, which occur every sixtieth of a second, update the cursor on the screen.

Figure 2-3. Layout of the System Static Variables Area



System Communications Area. The *System Communications Area* stretches from about $100 to $33F. It contains certain fundamental constants and variables that are shared among a number of system parts, as well as applications programs that run on the system. Because these quantities are shared by the rest of the system, they are often called *global*.

Certain locations in the System Communications Area specify the Macintosh's video screen. For example, locations $102 and $104 contain the vertical and horizontal resolution of the screen in dots per inch, and location $106 specifies the total number of dots in each row.

Other locations in the System Communications Area specify the memory layout. For example, location $108 (memtop) specifies the total amount of RAM in the machine, location $2B2 (RAMBase) contains the address of the beginning of the heap, location $2A6 (sysZone) contains the beginning address of the part of the heap used by the system, location $2AA (applZone) contains the beginning address of the part of the heap used by an applications program, location $114 (heapEnd) specifies the highest point in the heap, location $10C (bufPtr) specifies the highest part of regular RAM (excluding areas currently used for video, sound, and debugging), and location $2AE (ROMBase) contains the beginning address of the ROM.

Still other locations hold the time, date, and current state of the keyboard and mouse. To learn more about these, you can look through the source code for the assembly language library files, that come with Apple's development system for the Macintosh.

File System Globals. The *file system globals* are currently between $340 and $3FF. This area contains static variables used by the File Manager.

System Dispatch Table. The *System Dispatch Table* lies between $400 and $7FF. As we describe later, it is loaded with addresses of the ROM routines in a special compact form, and the Macintosh's Operating System uses it to find these routines.

The Mouse and Cursor System Globals. The *mouse and cursor system globals* currently go from $800 to $8FF. They can be thought of as QuickDraw's static variables. For example, location $824 contains the current beginning of screen RAM. This is one of the few locations referenced directly from ROM. But even then, its address is stored in a special section of ROM along with several other RAM addresses so that its location could be easily changed as the ROM was being developed.

Operating System Manager System Globals. Other *system globals* go from $900 to $97F. This section contains static variables for various managers in the Operating System, including the *Segment Loader*, the *Scrap Manager*, and the *Print Manager*.

ToolBox Global Variables. *Toolbox global variables* extend from $980 to $AFF. This section contains static variables used by the various managers in the Toolbox, such as the *Resource Manager*, the *Font Manager*, the *Window Manager*, the *Menu Manager*, the *Control Manager*, *Text Edit*, the *Dialog Manager*, and the *Package Manager*. We survey these managers at the end of this chapter.

The Heap

The *heap* currently begins at $B00 and extends upward toward the stack. As mentioned, the beginning address of the heap is contained in System Communications Area location $2B2 (RAMBase). The heap provides a place where dynamic data structures can be stored and moved as they change size. The heap stores Operating System routines, the Operating System's main program (called the "Finder"), the user's applications program, and many system and applications variables, including resources.

The heap is managed by the part of the Operating System called the *Memory Manager*, which has routines that can be called by other managers and by applications programs. It also performs some of its tasks in the background, responding to the vertical retrace interrupt, which occurs every sixtieth of a second.

Zones. The heap is divided into *zones* (see Figure 2-4). Each zone is separately managed by the Memory Manager. This allows memory to be "partitioned" for different uses (for example, *system* versus *application*). If the Macintosh were used as a multiuser system, then each user would have a separate zone.

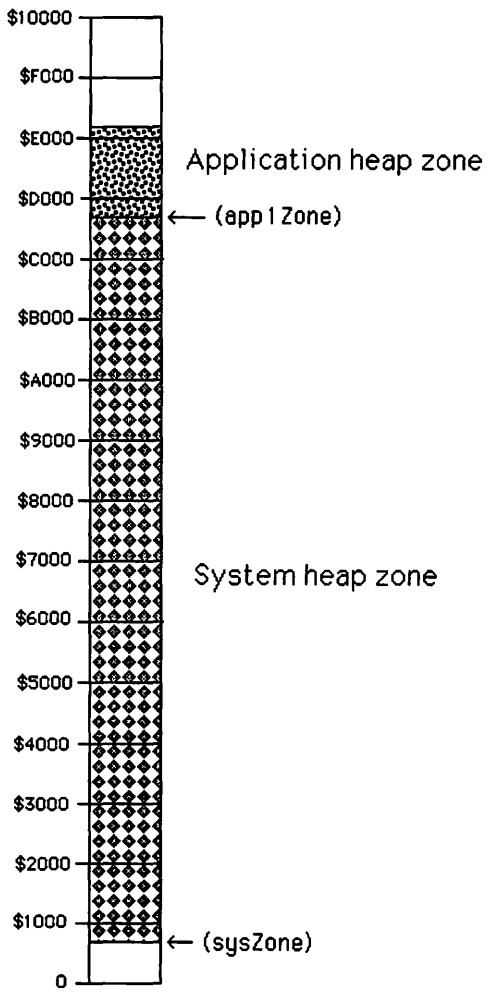
The first zone, devoted to the Operating System, is called the *system zone*. On a 128K Macintosh, it is normally 16.5K bytes long. On a 512K Macintosh, it is normally 48K bytes long. It contains code and data to run the "Finder" as well as such things as RAM routines that substitute for or supplement the ROM routines. Debuggers such as "MacNub", which are opened once a disk is booted, can also reside in this zone.

The second zone of the heap, called the *application zone*, contains the applications program and all dynamic data under its control. The minimum size is 6K, but the Memory Manager adds more room to the application zone in 1K increments as needed. However, the Memory Man-

ager cannot raise the top of this zone higher than within 1K of the lowest address of the stack (current top of stack). On a 128K Macintosh, this can prove a severe limitation for large programs with lots of data. When the Memory Manager runs out of room in this way, it tries to swap out parts of the applications program already in memory. This in turn can cause a good deal of disk activity, slowing down your program.

An application can call upon the Memory Manager to add more zones. Each zone begins with a special area of memory (about 52 bytes) called

Figure 2-4. The Heap Zones



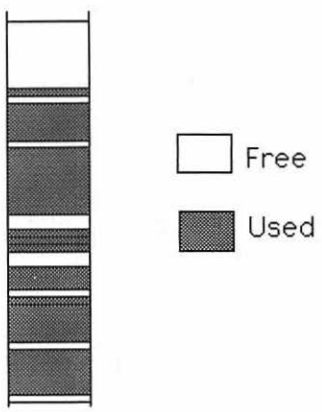
the *zone header*, which contains parameters that the Memory Manager needs to manage that zone. These include the addresses of key places within the zone.

After the header, a zone is divided into *blocks* that are dynamically allocated and deallocated within the zone as memory is needed for the individual data structures stored in that zone. The system heap may be divided into about thirty or more such blocks, one for each logical program or data structure. Each block contains a header that is used by the Memory Manager to size it and determine how it is being used.

In Chapter 3, we discuss how "pointers" and "handles" allow the Memory Manager to dynamically move data structures as they change size, yet still allow an application or other part of the system to properly access that data.

Fragmentation. As blocks are allocated and deallocated in a heap zone, "holes" develop (see Figure 2-5). These are blocks of unused (free) memory sandwiched between blocks that are in use. When more memory is needed in the zone, the Memory Manager tries to use these free blocks. Often, however, these blocks are too small to be used, and they remain as holes. As more holes develop, the memory begins to "fragment", with more and more storage wasted in the holes. The Memory Manager eventually tries to rearrange the blocks to remove the holes. This is called "memory compactification". The Memory Manager also tries to remove (purge) blocks that are not in use. To assist in this process, you should divide larger programs into *segments* and tell the Memory Manager when you no

Figure 2-5. Fragmentation



longer need a segment. Our example programs are small and thus do not need to be subdivided in this way. If your programs are large and require segmentation, you should consult Apple's *Inside the Apple Macintosh™* (Cupertino: Apple Computer, Inc., 1985).

Stack Area

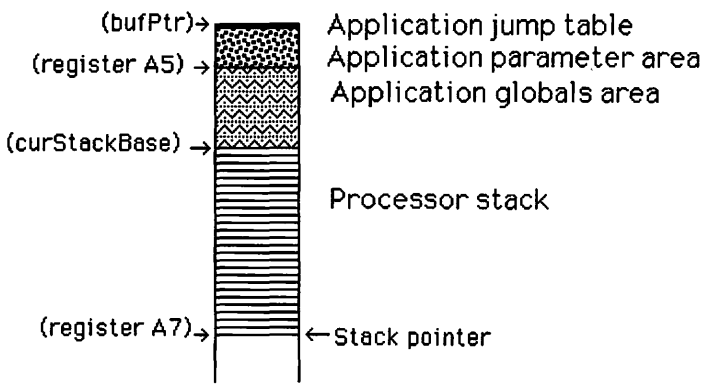
The *Stack Area* includes both the normal processor stack and some other areas of memory that act "stacklike". It starts at the address contained in location $10C (BufPtr) and grows downward toward the heap (see Figure 2-6). You can see in this figure the processor stack used by the 68000 CPU and the other parts of the Stack Area above it.

The stack represents an alternate approach to handling memory, different from how the system static variables or the dynamic heap work. It allows memory to be allocated and deallocated in a *sequential* manner. This is appropriate for a number of different quantities, such as return addresses and data for subroutines (the processor stack) and lists of variables used as global variables for an applications program.

To operate the stack, a special register called the stack pointer (register A7) always points to (contains the address of) the location where the last quantity was placed on the stack. The stack consists of all locations from the stack pointer upward to where the stack begins.

When new entries are added to the stack (see Figure 2-7) , the stack pointer is decremented by the appropriate amount according to the entry's size, and the new information is placed at this new location on the stack. This is called *pushing* entries onto the stack. Similarly, entries can be

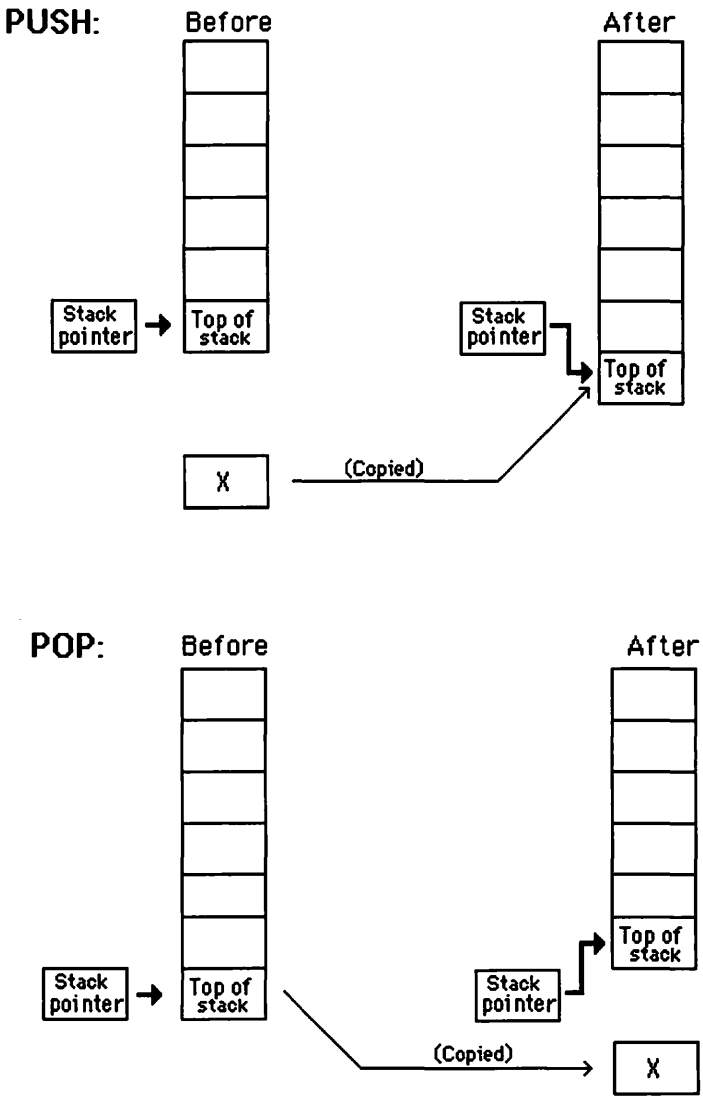
Figure 2-6. The Stack Area



popped off the stack by a reverse process. In this case the stack shrinks upward.

Once entries are put on the stack using the stack pointer (or some other method), other registers such as A5 and A6 can be used to read and even change their values. This last feature is not really included in the

Figure 2-7. Pushing and Popping the Stack



standard definition of a stack (which implies that you can access only the last stack entry). However, this is how the Macintosh and many other machines work, making the stack a much more valuable place to store things.

The *Application Jump Table*, the *Application Parameter Area*, and the *Application Globals Area* occupy the upper part of the Stack Area, above the regular processor stack. The *Application Jump Table* contains a minimum of eight bytes and sits just below the address contained in "bufPtr" (stored at location $10C). The *Application Parameter Area* contains 32 bytes and runs from the address contained in register A5 up to the Application Jump Table. The *Application Globals Area* runs from just above the address contained in "curStackBase" to just below the address contained in register A5.

The *Application Jump Table* contains information for referencing routines stored externally. This is needed when you divide your program into *segments*.

The *Application Parameter Area* contains parameters shared by the Operating System and the application, thus providing an *interface* between the system and the applications program. For example, when QuickDraw (the Screen Manager) is initialized, it sets up a path (series of pointers) through this area and then into your applications program. During normal operation, this path allows QuickDraw to use the application's drawing "environment" to determine how it will draw lines, text, and other shapes.

When a Pascal program is compiled, its VAR section (containing all its global variables) is packed into memory in the stack area. These variables are stored in a downward order because they are "pushed" into memory as they are compiled from your program. This is natural because some of the best compiler algorithms are stack-oriented.

When the system "launches" an applications program, it moves these stacks of variables toward the top of available memory (as specified by bufPtr). Once a program has been launched, the variables in these areas become static for the life of the program. On the other hand, a program's local variables are placed on the processor stack as the program executes. This provides a natural way to manage multiple copies of local variables that are needed for *recursion*, a valuable feature of Pascal that allows a procedure to call itself. Recursion methods are effective for such things as sorting.

Debugger Area

The area just under the video RAM can be used for debuggers such as "Macbug", which are installed during boot-up time. When these are installed, "BufPtr" is adjusted downward to point just below the debugger. Since the program launcher uses "BufPtr" to determine how much memory is available, any debugger installed in this manner remains safely tucked away, out of reach of the normal operation of the machine.

Video and Sound Areas

At the top of the RAM are areas that connect to the video and sound systems (see Figure 2-8). In many respects, they act as ordinary memory. However, they also connect to the screen and speaker.

Two areas of memory are devoted to the screen. On the 512K Macintosh, a primary area ranges from $7A700 to $7FC7F, and a secondary area runs from $72700 to $77C7F. These same values work with a 128K Macintosh because this smaller Macintosh ignores some of the upper address bits of RAM, "wrapping" the memory around so that larger addresses access the same memory cells as smaller addresses.

On the 128K machine, the "actual" addresses are $1A700 to $1FC7F for the primary area and $12700 to $17C7F for the secondary area.

Two areas of memory are also devoted to the sound system. Here are stored the "wave forms" for the sounds produced by the Macintosh. By controlling these bits, a programmer can produce a variety of custom sounds, just as a variety of pictures can be produced by controlling the bits of video memory.

Sound uses only the lower bytes of the 16-bit words in these areas. On the 512K Macintosh, the primary area ranges from $7FD00 to $7FFE3, and a secondary area runs from $7A100 to $7A3E3. Again, these same values work with a 128K Macintosh because the larger memory addresses "wrap around" on the smaller machine. On the 128K machine, the "actual" addresses are $1FD00 to $1FFE3 for the primary area and $1A100 to $1A3E3 for the secondary area.

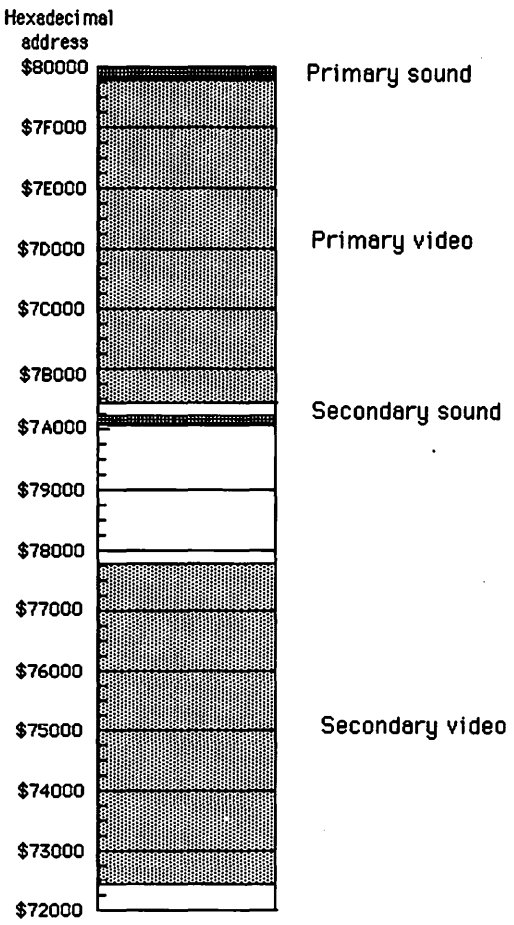
Bit 6 of location $EFFFFE is used to switch between the primary and secondary areas of the video and sound. This belongs to the VIA chip, which we discuss later. A value of zero selects the primary area, and a value of one selects the secondary area.

In Chapter 4 we examine how the video RAM works and how its memory "maps" to the screen.

The ROM

The ROM begins at $400000. This is at the one-quarter point of the 16-megabyte addressing space. In the last part of this chapter, we survey the built-in software that is stored in the ROM. In the rest of the book we explore it in detail.

Figure 2-8. Video and Sound Areas



Hardware Connections

The upper half of the addressing space from $800000 to $FFFFFF has a few scattered locations that are mapped to hardware devices. (See Figure 2-9 for a block diagram of the Macintosh.) These so-called memory locations actually access registers in the Macintosh's controller chips. Most applications do not need to, and should not, access these chips directly, but should use the Macintosh device drivers that are already developed.

Versatile Interface Adapter

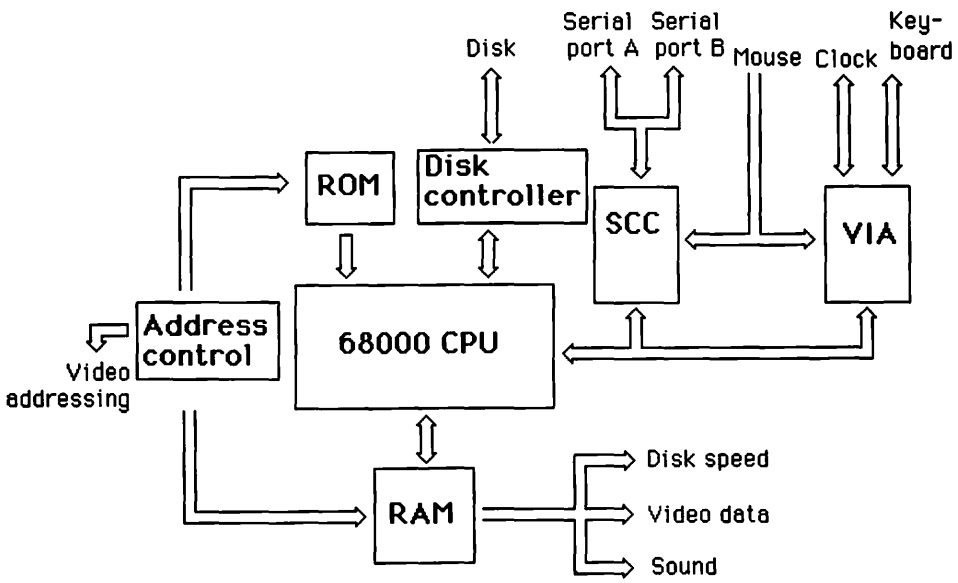
The *Versatile Interface Adapter* (VIA) controls access to the keyboard, real-time clock, and part of the disk, sound, video, and mouse.

The VIA uses locations in the range from $EFE1FE to $EFFFFE.

Serial Communications Controller

The *Serial Communications Controller* (SCC) controls access to the two serial communications lines. The Macintosh uses a Zilog Z8530 SCC chip, a powerful, dual-channel, high-speed communications controller capable of supporting all popular serial communications protocols.

Figure 2-9. Block Diagram of Macintosh



The serial interface chip uses locations in the range $9FFFF8 to $BFFFF9.

Disk Controller

The Macintosh uses two custom chips: one is called an Integrated Woz Machine (IWM) and the other a Microfloppy Controller Interface (MCI). It also uses an area of RAM to control the speed of the disk motor. This area is actually the upper bytes of the area in memory used for sound.

The IWM chip uses locations in the range from $DFE1FF to $DFFFFF. Signals from the IWM chip, the special area of RAM, and the VIA go to the MCI chip, which is part of the disk subsystem.

This completes our survey of the Macintosh's memory usage.

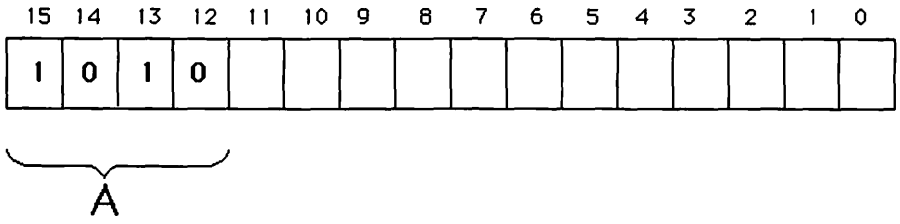
Accessing the Macintosh Built-in Software

Now let's study the ingenious method that Apple uses for providing access to its built-in software. This method substitutes routines in Macintosh's memory for certain unimplemented instructions of its MC68000 microprocessor, thus extending the original set of processor instructions by a whole new set of "Macintosh" instructions. In this section, we explain how this works.

The instruction codes for the Macintosh's 68000 processor are 16-bit integers. However, many of the 65,536 possible values do not belong to any processor instruction. These are called *unimplemented instruction codes*.

Two special families of unimplemented instruction codes are indicated by their upper four bits (see Figure 2-10). A bit pattern equal to 1010 in these bit positions indicates the family used by the Macintosh to

Figure 2-10. The 1010 Family of Unimplemented Instructions



access its own routines. In hexadecimal notation, the bit pattern 1010 corresponds to the digit "A". Thus, these codes are distinguished by a leftmost hexadecimal digit equal to a value of "A". A second family of unimplemented instruction codes, distinguished by a bit pattern 1111 in the leftmost bit positions, is unused by the Macintosh. Perhaps a clever programmer will use this family to implement an entirely new set of custom instructions.

Whenever the processor encounters such a 1010 instruction code, it does a special interrupt called the 1010 unimplemented instruction interrupt, which calls the dispatch routine. The exception vector for this interrupt is at location $28.

The service routine that Apple installed to handle this particular interrupt branches to the appropriate Macintosh routine. To make this *dispatcher* routine work correctly, Apple has installed a table called the *System Dispatch Table* in the Macintosh's low RAM area that contains the addresses (in compressed format) of all the Macintosh's routines that are accessed in this manner.

When the machine is turned on or restarted, this table is loaded from ROM and "unpacked" into low RAM (starting at address $400 and running to address $7FF). At this point, all addresses in the table supposedly point to ROM routines, although this is hard to verify because the machine does not do much except wait for a disk insertion when it starts up. When a disk is inserted, a few new routines are loaded and a few entries in this table are overwritten with address information for the new routines. Many of these new routines simply do a few extra things and then jump to the original ROM routine, but a few are complete replacements. This provides a convenient way to change the system as needed.

Each entry in the address table is a 16-bit integer (see Figure 2-11) that is expanded by the dispatcher routine in the following manner. Bit position fifteen distinguishes between the built-in ROM routines and the additional RAM routines, and the lower 15-bit positions give the *word* offset for the routine. For ROM routines (bit 15 equals zero), the *word* offset is multiplied by two and added to the base address of the ROM ($40000) to get the *byte* address of its entry point. For RAM routines (bit 15 equals one), the *word* is multiplied by two and added to the beginning of the heap ($B00) to give the *byte* address.

About 494 routines are handled in this manner. Each routine is assigned a unique nine-bit instruction *number* between 0 and 511 corresponding to its position in the dispatch table. About eighteen numbers scattered throughout this range are not used, and about twelve other numbers are assigned names but are not documented at this time.

The 16-bit operation *codes* for these instructions are computed by placing the special 1010 bit pattern in the upper four bits (bits 13 through 15) and the instruction number in the lower nine (bits 0 through 8). This leaves three bits in the middle for other purposes, such as setting certain instruction modes.

To complicate the situation, the routines are divided into two classes: *Operating System routines* and *Toolbox routines*. Generally, Operating System routines are assigned instruction numbers between 0 and 255 and have bit 11 equal to zero, and Toolbox routines are assigned instruction numbers within the full range 0 through 511 and have bit 11 equal to 1 (see Figure 2-12).

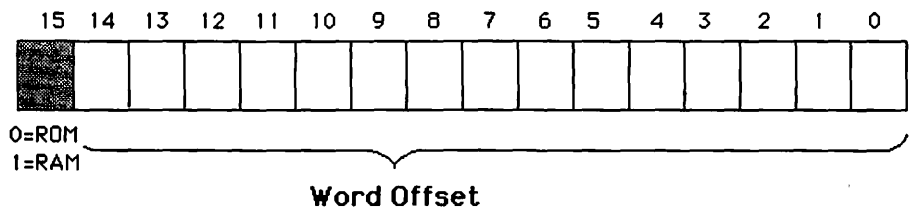
For Operating System routines, bits 9 and 10 (called flag bits) send special information that depends on the particular routine, and bit 8 indicates whether or not register A0 is being used to pass information back from the routine. Again, this depends on the particular routine.

For Toolbox routines, bits 9 and 10 are not used. Bit 10 was once used to distinguish a special "auto pop" mode for toolbox routines but is now unused.

Notice that bit 8, part of the instruction number for Toolbox routines, indicates the parameter passing mode for Operating System routines, as described previously.

Nice distinctions between Operating System and Toolbox are not always adhered to. Routines with instruction numbers 0 through 4F form the main part of the Operating System and follow all rules for Operating System routines. However, some routines that logically belong to the Operating System behave as though they were Toolbox routines. That is, they have bit 11 turned on and may have instruction numbers greater than 255.

Figure 2-11. System Dispatch Table Entries



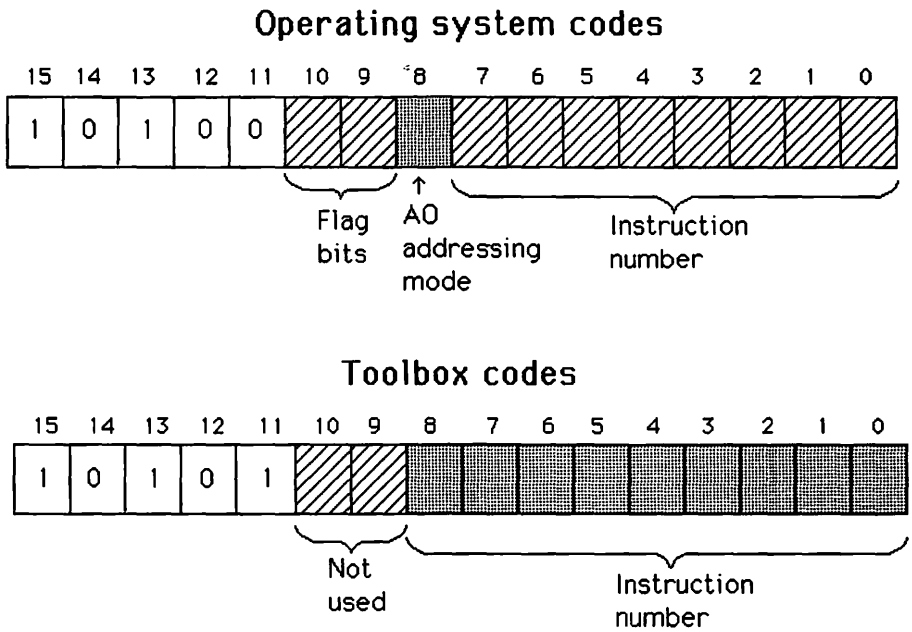
The Managers

Now that you have an idea of the physical organization of the Macintosh's memory and instructions, you are ready for a first look at the software that occupies this memory space.

As mentioned before, the Macintosh's built-in software is logically divided into modules called *managers*. Each manager has a specific area of responsibility in the system. For example, the Memory Manager manages the heap, the Window Manager maintains various windows on the Macintosh screen (see Figure 2-13), and the File Manager is in charge of the file system. These managers work together to run the Macintosh just as people in an office work together to run a company.

Some managers are considered part of the Operating System, and some are considered part of the Toolbox (see Table 2-1). However, this distinction is blurred and somewhat artificial. Generally, managers that relate to the operation of the screen are considered part of the Toolbox, and managers that manage other parts of the system such as files and memory are considered part of the Operating System.

Figure 2-12. Operating System and Toolbox Instruction Codes



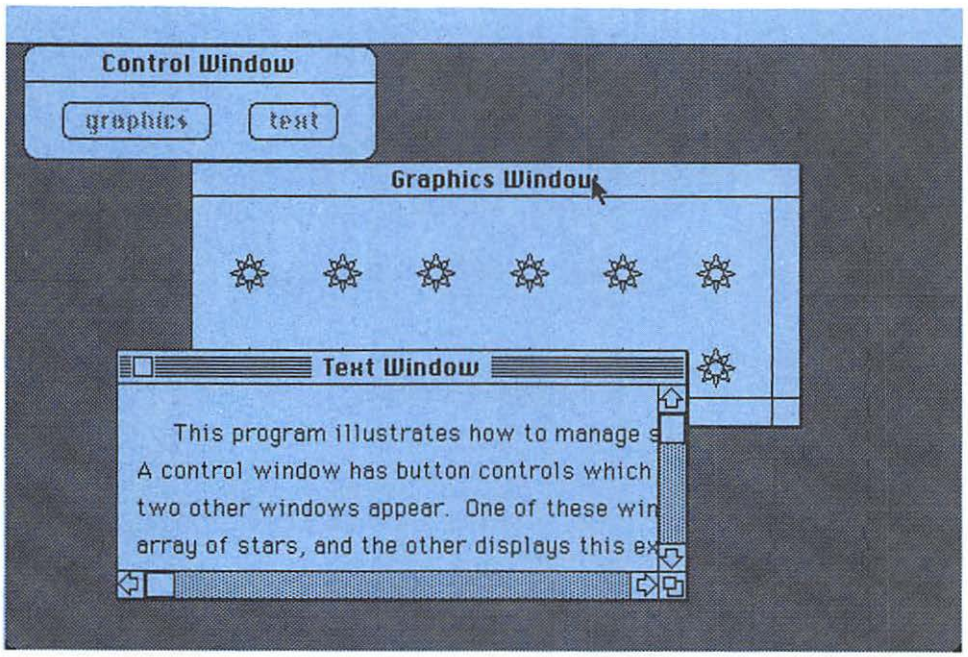
Dividing the total system into managerlike pieces is a well-established practice to increase the speed of software development. This allows several members of a software development team to divide responsibilities, then smoothly bring their efforts together. It also makes further maintenance development of the system easier and makes it much easier for an applications programmer to understand the system.

Physically, these managers consist of routines mostly in ROM, but partly in RAM; and data structures in RAM. ROM routines belonging to a specific manager tend to be grouped together. However, some are intermixed with routines from other managers. This intermixing occurs even in the dispatch table.

Only at the software interface level are routines clearly organized by manager. Even at this level, distinctions blur. For example, the file system "open" routine is shared by the File Manager and the Device Manager, and string routines are scattered among QuickDraw, the Operating System utilities, the Resource Manager, and the Package Manager.

In addition to managers, there are two collections of routines called utilities. The Toolbox utility routines do such tasks as bit manipulation.

Figure 2-13. Macintosh Windows



The Operating System utility routines do such things as read and set the time and make a beeping sound on the speaker.

In this section, we survey these managers and describe their most notable and useful features. In later chapters, we examine their operation in detail.

The Screen Manager

QuickDraw is the name of the set of built-in routines and data structures that directly manage the screen. It is considered part of the Toolbox.

The QuickDraw routines control the drawing environment by setting certain drawing parameters, such as the size and position of the drawing area, the size and pattern for the pen, and the size and style of text.

QuickDraw also contains routines to move the pen around the drawing area, laying down lines with the various pen sizes and pen patterns. Other QuickDraw routines draw text in various sizes and styles. Others control the shape and appearance of the cursor.

Still other QuickDraw routines draw a variety of shapes, such as lines, rectangles, ovals, rounded rectangles, polygons, and irregularly shaped areas called regions. With most of these shapes, there are the options of outlining (framing), filling, inverting, or erasing.

Table 2-1. The Managers (in ROM)

Toolbox	*Operating System*
QuickDraw (the Screen Manager)	Event Manager
Toolbox Utilities	O.S. Utilities
Font Manager	Memory Manager
Event Manager	File Manager
Resource Manager	
Window Manager	Segment Loader
Control Manager	Vertical Retrace Manager
Dialog Manager	
Menu Manager	
Desk Manager	
Text Edit (the Edit Manager)	
Scrap Manager	
Package Manager	

Routines also detect when points and certain shapes are within other shapes. Other QuickDraw capabilities include storing and later replaying entire sequences of QuickDraw routines.

The Font Manager

Associated with QuickDraw is the *Font Manager*, which controls the storage of various character sets. It is considered part of the Toolbox. The Font Manager has routines that allow you to select from available character sets. QuickDraw does the actual drawing of text.

The Memory Manager

The *Memory Manager* controls dynamic memory allocation. It is considered part of the Operating System.

The Memory Manager routines set up and control dynamic variables that are used by the Operating System, the Toolbox, or an applications program. These variables are stored in the *heap* and accessed through pointers and handles, which are described in Chapter 3.

The Memory Manager routines are called by many of the other managers when they need to store or manipulate their data structures. The Memory Manager also works in the background.

The Event Manager

The *Event Manager* controls the interactive nature of the Macintosh. As described in Chapter 5, part of the Event Manager resides in the Operating System and part resides in the Toolbox.

The routines in this manager monitor hardware, such as the keyboard, mouse, screen, and disk. When these devices indicate user actions (events) such as key or mouse button presses, the Event Manager places the relevant information, including the nature of the event and the current time and position of the mouse, on an *event queue* (waiting line). The programmer can then get these events in an orderly manner by calling a special Event Manager routine.

The Event Manager also contains routines to directly access the condition of the mouse button and keyboard.

The Window Manager

The *Window Manager* controls the various Macintosh windows that are set up by the Operating System or by an applications program. It is considered part of the Toolbox.

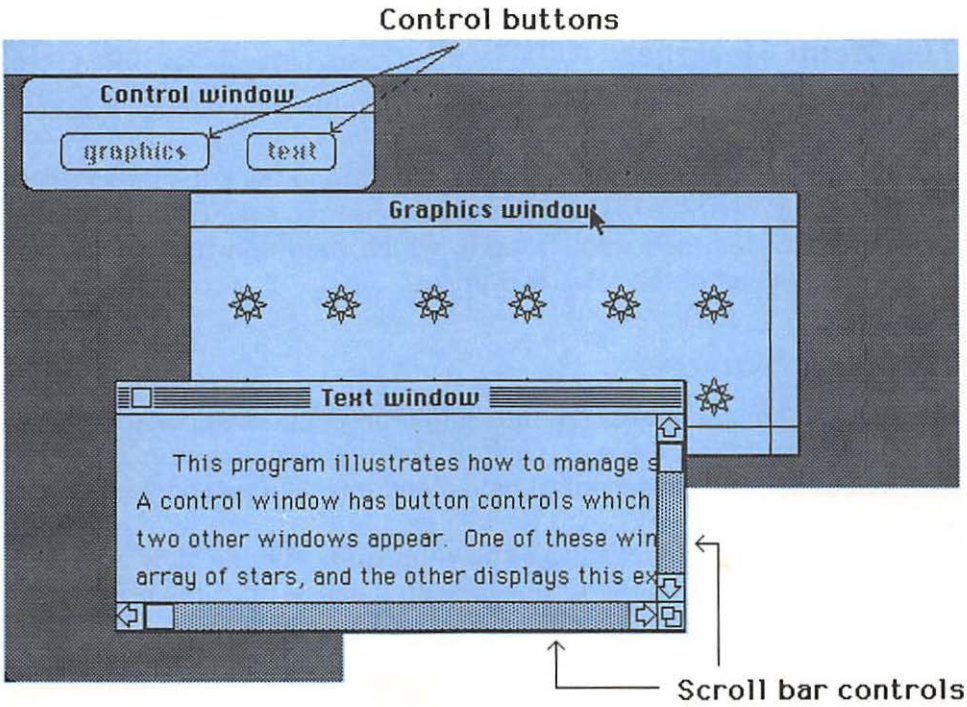
The Window Manager routines define the size, position, and features of a window. This Manager has routines to draw, redraw, highlight, and erase windows, performing the appropriate actions as several overlapping windows are maintained on the screen at once. Other routines move and resize windows on the screen in response to mouse movements.

The Control Manager

The *Control Manager* maintains the various buttons, check boxes, dials, and scroll bars that appear within Macintosh windows (see Figure 2-14). It is considered part of the Toolbox.

Associated with each control is a numerical value called its *control value*. The purpose of any control is to allow the user to manipulate that

Figure 2-14. Macintosh Controls



value. For example, each scroll bar control allows the user to control the horizontal or vertical displacement of the page.

The Control Manager has routines to define the sizes, positions, and features of controls; routines to draw, redraw, highlight, and erase controls; routines to track the mouse as it interacts with controls; and routines to set and delimit the numerical value associated with each control.

Tracking can be automatic or custom designed by the applications programmer.

The Dialog Manager

The *Dialog Manager* is a higher-level window manager, controlling entire user interactions without the intervention of the applications programmer. It is considered part of the Toolbox.

The routines in the Dialog Manager define the size, position, features, and controls in a dialog; start up and end a dialog; transfer information in and out of a dialog; and allow editing of dialog text items.

Routines also start up *alerts*, which reside at an even higher level than normal dialogs in that the programmer does much less to make these work. The programmer need call only one routine which handles the entire process.

The Menu Manager

The *Menu Manager* controls the Macintosh's pull-down menus. It is considered part of the Toolbox.

The Menu Manager routines define menus and track the menu selection process. With these routines, a programmer can specify the name and styles of each item and know exactly which item of which menu was selected at the end of the selection process.

The Desk Manager

The *Desk Manager* controls the interaction between an applications program and the desk accessories, such as the scrapbook, alarm clock, note pad, calculator, key caps, control panel, and puzzle. It is considered part of the Toolbox.

The routines in this manager allow the application to start up a desk accessory and give it slices of time to keep it active until the user tells it to close.

The File Manager

The *File Manager* provides access to the files on the disks and the serial communications lines. It is considered part of the Operating System.

With the routines in the File Manager, an applications programmer can do all the usual operations with files, including create, open, close, read, write, rename, delete, get, and change file information such as file types and attributes. This Manager also has routines to mount, unmount, and eject disks and select files.

Text Edit

Text Edit is the manager that handles blocks of text being edited. It is considered part of the Toolbox.

The Text Edit routines perform such functions as allocating and deallocating space in memory for text, formatting text for display, drawing and redrawing text, maintaining the blinking cursor or inverted selection range, inserting characters, scrolling text, and handling the usual editing functions such as cut, copy, and paste.

Packages

The *Package Manager* provides access to system routines into the system that are not included in the ROM. There can be as many as eight different "packages", and each package can contain a large number (as many as 64K) of different routines. Currently, the largest number of routines in any package is nine.

Some examples are a package of routines to help initialize disks, a package of routines to help select files, and a package of routines to handle time, dates, and strings according to various international rules.

The Resource Manager

The *Resource Manager* provides the applications programmer with access to an application's *resources*. It is considered part of the Toolbox.

A Macintosh applications program consists of a collection of *resources* (see Figure 2-15). One type of resource is the program *code*. Others include definitions of various objects managed by other managers. For example, the size, shape, and features of each *window* and *control* are normally stored in a separate resource.

As shown in Chapter 3, part of the development process for an applications program consists of packaging its various resources into a *file* on

a Macintosh disk. Except for the code resource, resources are normally specified in what is called a *Resource Definition File*, which acts like source code for the various resources.

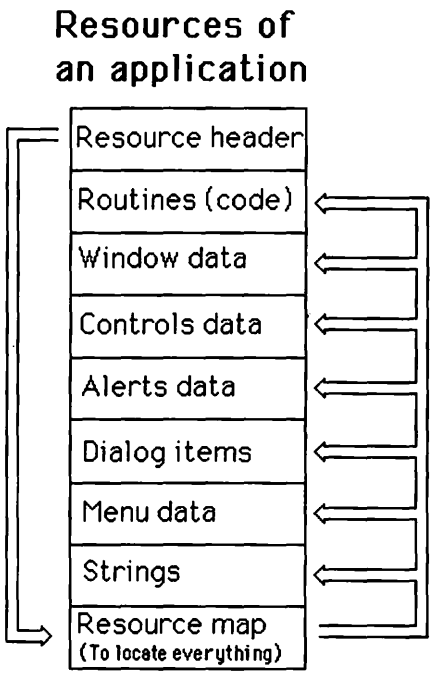
Besides the resources packaged with an application, the Macintosh maintains other resources that can be accessed by the applications program.

Much of the time, the Resource Manager stays behind the scenes, providing support for other managers when they need to get resources. For example, "GetNewWindow" is a Window Manager routine that calls upon the Resource Manager to get the parameters to define a new window.

The Scrap Manager

The *Scrap Manager* manages the clipboard to cut, copy, and paste. Clipboard routines need to be called if an application transmits information to and from other parts of the system, such as desk accessories and other applications.

Figure 2-15. A Program is a Collection of Resources



Other Managers

The Macintosh contains other managers. Some are directly used by an applications program, some normally stay behind the scenes. Of course, a sufficiently adventuresome applications program could use some or all of these managers.

In the Operating System

Other managers in the Operating System include the *Vertical Retrace Manager*, the *Device Manager*, and the *Printing Manager*.

The *Vertical Retrace Manager* is in charge of updating the system every sixtieth of a second. At this time, the video system finishes a complete scan of the screen and gets ready for the next scan. The screen is not written to during this "retrace", allowing the screen memory to be updated with fewer side effects. It is also a convenient time to update the system's time, check the stack size, check if a disk has been inserted, and move the cursor. An applications programmer may also call the Vertical Retrace Manager to install custom routines to be performed on a frequent and regular basis.

The *Device Manager* provides access to devices such as the serial ports as though they were files with special control and status functions.

The *Printing Manager* handles printing. It does not reside in ROM but is brought into RAM as needed.

Other Parts of the System

The Operating System contains a number of lower-level parts, including the dispatcher routine for Macintosh routines and *device drivers*.

Device drivers form the interface between the higher-level managers and memory locations that control and pass information to devices such as the serial communications lines, the sound system, and the disk system. An applications program can talk directly to these drivers, but for the most part this should not be necessary.

Summary

In this chapter, we have described the overall internal structure of the Macintosh, starting with its physical layout and leading to its logical structure as a collection of managers. We study these managers in detail in subsequent chapters.

3

Programming the Macintosh

This chapter covers the following new concepts:

- **The Program Development Environment**
- **Source Code**
- **Library Files**
- **Resource Definition Files**
- **Pascal Pointers**
- **Static and Dynamic Variables**
- **Relocatable and Nonrelocatable Areas of Memory**
- **Heaps and Stacks**
- **Debugging**

This chapter introduces a standard programming environment for developing applications programs for the Macintosh. This programming environment allows the user to develop programs such as spreadsheets, editors, and file utilities, making the Macintosh a powerful information handling tool.

We explain why this particular environment was chosen and describe how the example programs in this book are developed. We present an example program and discuss how it is written and processed as a finished application that runs on the Macintosh. Although this section describes

a program development environment, it emphasizes the basic functions that must be performed in any development system. These basic functions will be needed in any future program development environment. Thus, this discussion is valuable even when using a different development process.

In this chapter, we introduce the idea of *resources*. Macintosh applications program development differs from most others in that its programs are packaged as collections of resources. An applications program uses these resources to perform its job. We explain this concept and show how the program code is one such resource in the package.

We also explain how Pascal *pointers* work. Apple Pascal's implementation of pointers makes the language a very powerful tool for controlling the Macintosh's hardware while maintaining a modern, structured programming environment in which large programs can be easily developed and maintained.

We describe how to set up a debugger that allows us to see, and therefore fully understand, how features such as pointers work on the Macintosh.

Programming Environment

This section presents the programming environment used to develop the example programs in this book. We discuss the basic hardware and choice of programming language.

In the next section, we describe the program development process in detail, presenting a simple example program.

Choice of Hardware

The preferred methods for developing programs will change as more software tools are developed and the Macintosh's hardware becomes more powerful with larger main memory and larger, faster secondary storage (for example, well-integrated hard disks). However, as of this writing, efficient program development requires a Lisa (now called Macintosh XL) to run Pascal as well as two Macintoshes. In view of this evolution, we describe the process in general terms, mentioning the current hardware environment merely to illustrate the discussion. Thus no matter what system you use, you can gainfully read this chapter, substituting the details of your own system for the Lisa Pascal system.

Currently, program files are written and processed on a Lisa (Macintosh XL). The resulting application is then transferred to and run on a regular Macintosh. During program development, a second Macintosh

displays debugging information about the first Macintosh and controls the program. The finished program runs on a stand-alone Macintosh.

Traditionally, applications for a new computer are first developed on an older, larger machine that already has the proper development tools written for it. Some large software development firms continue to use large computers to develop applications, even after a new machine is well established. However, many companies and individuals that cannot afford large computers use the "target" machine itself to develop applications.

The Lisa is not so different from the Macintosh itself. In fact, Apple is in the process of merging it with the Macintosh. With the proper amount of main memory, fast disk storage, and a few changes in the Operating System, a basic Macintosh could actually be more powerful than the older Lisa (Macintosh XL), thus becoming a suitable environment for its own program development. Maybe by the time you read this book, Macintosh applications will be developed directly on a hard disk version of the basic Macintosh.

Already, there are debuggers for the Macintosh that require only one machine, allowing you to flip back and forth between a debugging screen and the normal output. However, they will never be quite as good as a two-machine debugging system, since it is always useful to completely separate normal program input and output from that of the debugger.

A two-machine Macintosh system is not unreasonably costly when considering the benefits, but it is desirable to eliminate the Lisa when its function can be taken over by one of the Macintoshes. The same kinds of tools should then be available for such a Macintosh.

Why Use Pascal?

The examples in this book are written in Pascal. Even if you don't know Pascal, you should find these examples easy to understand, and even if you don't plan to write your applications in Pascal, you will learn some very important lessons from these examples. Although the Pascal language is used here merely to explore the Macintosh's programming environment, the Apple version of Pascal is actually a good choice as a development language.

First, let's discuss the advantages of Pascal. The primary advantage is that Pascal is the original development language for the Lisa and the Macintosh. Assembly language has also been used, but in a supplementary role to Pascal. Thus, the tools for Pascal are well developed and the basic structure of the Macintosh is oriented towards Pascal. Apple's documentation is also written in Pascal; that is, even though it is written in English, it uses Pascal contructs to explain the Macintosh's operations.

A second advantage is that Pascal is one of the first widely accepted structured languages; that is, it uses widely accepted program and data structures. Pascal was developed as a result of researching how such structures can and should make programming easier and more reliable.

Pascal's structures appear to be "generic": its basic structures are so appropriate and necessary that they have been borrowed by other languages, such as BASIC, FORTRAN, and some assembly languages. Although these languages have been around longer than Pascal, new versions of them have been developed that incorporate Pascal-like structures. This tends to make Pascal easy to read and understand, even for those who don't know the language but know modern versions of these other languages.

Pascal-like structures are essential to good programming practices, allowing programs to be developed in a systematic manner so that the overall organization and structure of each component of the program can clearly be seen. It allows *procedures* (subroutines) to be written and given names that spell out their function. These names are then added to the vocabulary of things that your program can do. This can be done very effectively in a hierarchical manner so that each part of your program looks like an outline of what it is accomplishing.

Data structures can also be organized hierarchically so that a larger structure can be referenced as a whole, but each piece can also be easily accessed.

Apple Pascal

Apple's implementation of Pascal has a number of advantages. It is a higher-level language that translates directly to the machine language used by the Macintosh's 68000 central processor. When we discuss debugging, you will see how to display the resulting machine language in the form of 68000 assembly language in the windows of the debugging screen. You can then easily, interactively explore any of our example programs in assembly language.

A consequence of Pascal's generic nature is the need for extensions. Of course, because of its hierarchical nature, it is self-extending; one can write packages, as Apple has done, that extend Pascal so that it conforms to a particular environment. However, basic elements such as dynamic strings are missing from standard Pascal. Fortunately, Apple Pascal has a nice extension to handle dynamic strings.

Apple Pascal adds a number of features to an interesting set of Pascal structures called *pointers*. Pointers are essential to the operation of the Macintosh. Later, we discuss pointers and the special operations for them added by Apple. With these facilities Apple Pascal becomes a very pow-

erful language that allows us to directly control the machine hardware while maintaining the advantages of a modern, structured, higher-level language.

Perhaps a word of warning is in order. A few features in Pascal are not well defined. Different versions of Pascal may implement these features differently. For example, some *data typing* does not clearly define how it is packed into memory. In particular, the "subrange" of numbers "0...255" may be interpreted as a byte-sized piece of data in one implementation of Pascal (the version we use for these programs) or as a subrange of 16-bit integers in another Pascal (the interpreted Pascal that runs directly on the Macintosh). However, this type of problem is really at the implementation level, since it relates to the way memory is implemented on the computer.

The Development Process

In this section, we describe the development process in broad terms, explaining the *logical* generic rather than the specific *physical* steps. This should help put the example programs into perspective and allow you to implement them on your own system. In particular, this section describes *resources*.

As mentioned above, the exact steps vary that are required to put an application together. If you need a more detailed description of the current development process, please read Appendix B.

We illustrate the development process with listings of a trivial program. The program is not designed to be useful; it merely illustrates editing, compiling, linking, loading, and running — processes used in all the example programs in this book. By studying this program, you will understand how the other example programs as well as your own applications can be implemented on the Macintosh.

Source Files

The development process begins with *source files*. These textual "documents" are written and modified using an *editor* and stored on some mass storage device such as a disk. They can range in size from a few characters (or even zero) to several thousand characters. The files for this book are written and stored on a Lisa. However, you can write such files directly on the Macintosh or on a different computer altogether.

Because of the intrinsic nature of Macintosh applications programs, two source files are needed: one contains *source code*, the other contains *resource definitions*. Currently, these are in separate files, but other systems could possibly combine both types of information into one file.

The *source code* file contains the *Pascal program*. This is compiled into MC68000 *machine language* and linked with other machine language modules to form a complete machine language program. Later, we look at the Pascal source code file and associated files, called library files, that are supplied by Apple.

Macintosh Resources

First let's look at the *resource definition* file. To understand why this file is necessary and how it works, you should know something about the structure of applications as they sit upon a Macintosh disk.

All files residing on a Macintosh disk consist of two parts — a *data fork* and a *resource fork*. Both parts must be present, but either part can be empty.

The *data fork* contains normal user data, such as text in a text file or data from a spreadsheet program. In Chapter 10, we see how to open, read from, write to, and close the data forks of Macintosh files, like ordinary files on most computer systems.

The *resource fork* contains the program. Officially, it contains a number of items called *resources*, but the primary resource is the 68000 machine code and data that comprise your compiled program. Other resources might define the basic parameters for *windows*, *dialogs*, and *text* that are used by your program. We study these resources in later chapters. Table 3-1 lists the common types of resources.

Currently, Apple supplies a program called a *resource compiler* (called "RMaker"), which converts your *resource definition file* along with your compiled program into a file that contains your finished application, ready to be run on the Macintosh in stand-alone mode. Currently, this finished file must be transferred from the Lisa to the Macintosh before it can be run.

Apple is also developing a *resource editor* that allows a programmer to add, remove, and modify resources in a program. Each type of resource can be edited in its own manner. For example, icons, patterns, cursors, and fonts can be edited in enlarged forms just like "fat bits" in MacPaint.

If we think of an application as a collection of resources, then we should think of the *resource definition file* as the main file for defining our application because it specifies the application's resources (see Figure 3-1).

In the example program for this chapter, the compiled program is the only resource. In later chapters, we gradually introduce other types of resources and the relevant concepts behind them.

Now let's examine the resource definition file for our example. Notice that the resource definition file does not actually contain the code resource, but names the object file where that code may be found.

```
* resource file for Trivial demonstration program
clm/Trivial.rsrc

Type CODE
clm/TrivialL,0
```

The first line is a comment. All comments in a resource definition file begin with an asterisk. You can have any number of lines of comments at the beginning. However, you must be careful about putting comments in other parts of the file because they might be interpreted as resource definition information.

The second line of this file declares the name of the file where the finished application will be placed. Currently, this is a Lisa file, which is then moved by a file transfer program called "MacCom", onto a Macintosh

Table 3-1. Common Types of Resources

Type	*Object Described*
WIND	Window
MENU	Menu
CNTL	Control
ALERT	Alert
DLOG	Dialog
DITL	List of items in an Alert or Dialog
ICON	Icon
ICN#	List of Icons
CURS	Cursor
PAT	Pattern
Pat#	List of Patterns
STR	String of text
STR#	List of strings
DRVR	Desk Accessory
FREF	File reference
BNDL	Bundle
FONT	Font
CODE	Machine code for a program

disk that is inserted into the Lisa disk drive. In the future, this may be your finished application file on the Macintosh.

The third line is blank simply to make the file more readable. Although some lines in a resource definition file must be blank, this one does not have to.

The fourth and fifth lines specify the code resource to be included in the application. This is the compiled and linked program.

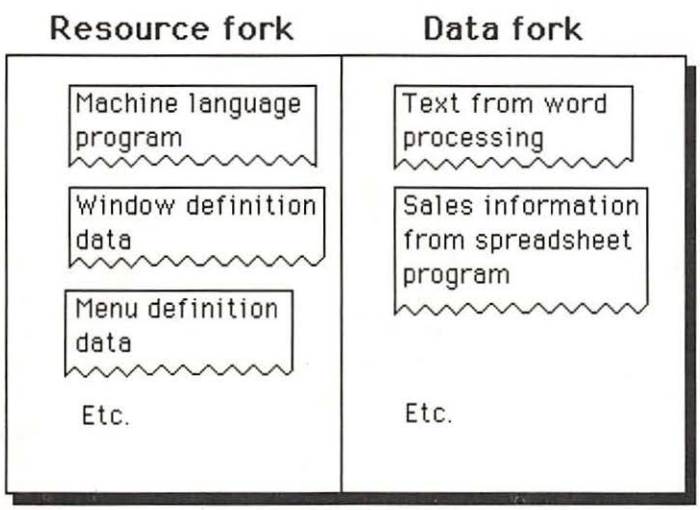
The fourth line is a TYPE statement that declares the resource to be of type CODE. We encounter other types of resources later. The fifth line specifies a file where the machine code is found. In this case, the file is "clm/TrivialL", which is the compiled and linked version of our Pascal program.

Pascal Source File

Let's now look at the *source* file for this trivial program.

```
PROGRAM Trivial;
   { $R-}{$X-}
```

Figure 3-1. Resources and Data in Files



```
USES
   {$U obj/Memtypes    }Memtypes,
   {$U obj/QuickDraw   }QuickDraw,
   {$U obj/OSIntf      }OSIntf,
   {$U obj/ToolIntf    }ToolIntf;

BEGIN
  {This program does nothing}
END.
```

Program Statement

The first line of this program is the *program statement* (as Pascal programmers know). This statement specifies the name of the program: in this case, "Trivial".

Compiler Commands

The second line of the program contains two compiler commands. A *compiler command* is an instruction to the compiler that is not part of the Pascal language.

These compiler commands ensure that the program runs properly on the Macintosh. Note that the same Pascal compiler can compile programs that run on the Lisa.

Each compiler command starts with a dollar sign and is enclosed in brackets as a Pascal comment. This hides them as far as the Pascal language is concerned but allows the Pascal compiler to easily recognize them.

The first compiler command is $R-, which disables range checking in your program. Range checking verifies whether variables such as array indices are within designated bounds. The current version of the compiler does this incorrectly and may cause a program to crash. Since the default setting is $R+, which enables range checking, we need $R- to disable this particular feature. Future versions of the compiler should have this bug fixed.

The second compiler command is $X-. This command disables automatic stack expansion. Automatic stack expansion is desirable for programs that run on a Lisa; however, this feature is not appropriate for Macintosh programs because the Macintosh has a different way of handling memory. The default setting is $X+, which enables automatic stack expansion; thus, we need the $X- command to turn this feature off.

Apple also recommends that you invoke the $U- command to disable the use of Lisa library files. The Macintosh library files, discussed below, invoke this command for you. It is therefore not really necessary to issue this command from your program.

External Files

The USES section of this program allows our program to take advantage of the large number of Pascal declarations developed by Apple that provide access to the Macintosh's ROM.

These definitions are stored in a number of compiled external Pascal files called library files. These are the *physical* containers of this information. Each library file can contain one or more *logical* UNITs. A UNIT is a special module that is part of the Apple Pascal language. In Chapter 4, we discuss the contents and structure of UNITs in detail.

The $U compiler command connects the *file name* to the *UNIT* names. It causes the compiler to search the specified file for all UNITs mentioned subsequently in the USES section until the next $U command.

In this program, we use the following external library files: "MemTypes", "QuickDraw", "OSIntf", and "ToolIntf" (all prefixed by an "obj/"). These library files contain only one UNIT. The name of the UNIT for these files is the same as the name of the file that contains it (ignoring the "obj/" prefix to the file name). In each case, a U$ compiler command enclosed within Pascal comment brackets gives the file name, thus making it not part of Pascal. Immediately after, the corresponding UNIT name is given as part of the USES statement in Pascal.

In Chapter 4, we discuss the structure and contents of UNITs. For now, understand that they give you extra data structures and procedures that are not a regular part of Pascal.

Main Program

The main program consists of a BEGIN and an END statement. This is the absolutely smallest main program in Pascal. As you can see, the program does no useful work. It merely signs on, then off.

Putting It All Together

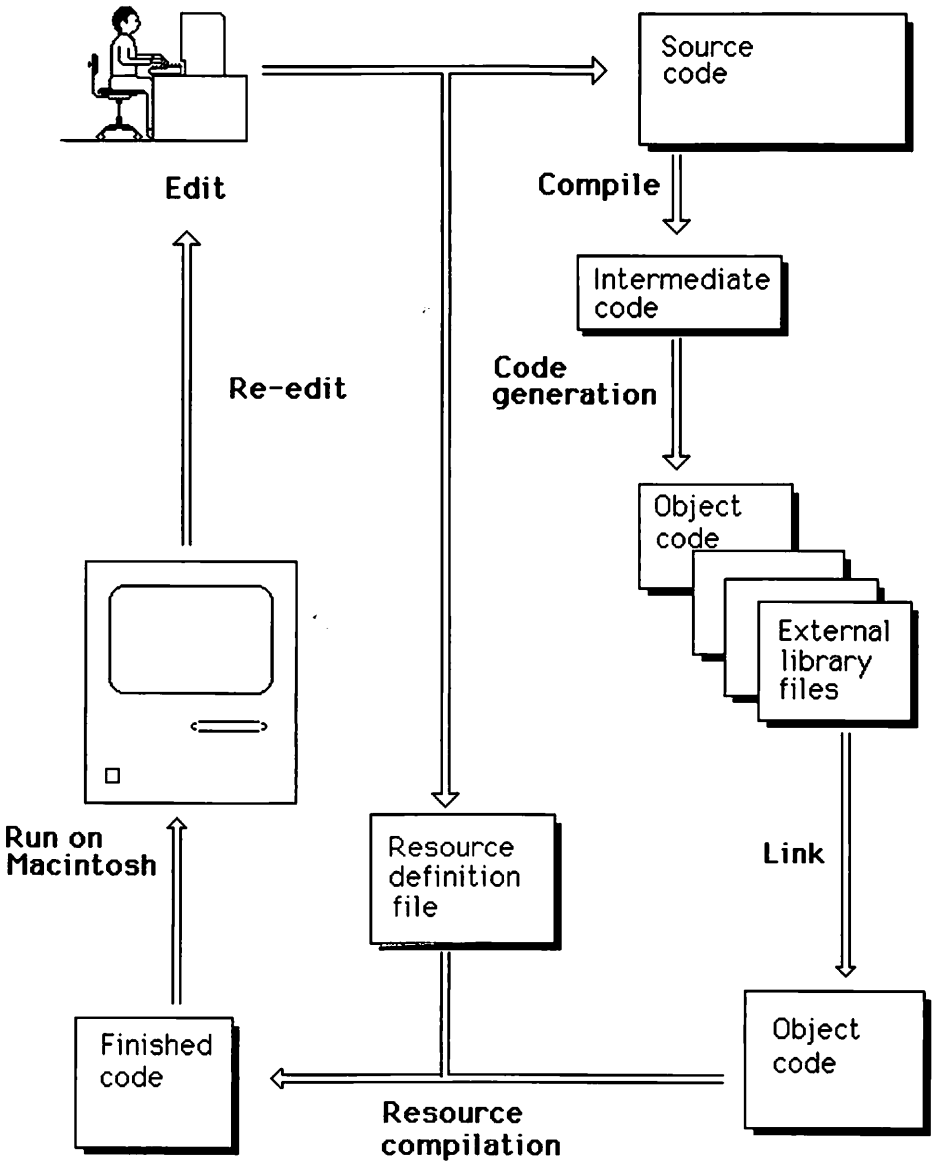
The Pascal source code file needs to be compiled and linked, then combined with the other resources to form the complete application (see Figure 3-2). We describe this process only briefly here. Details of the current steps are in Appendix B.

Currently, compiling takes two steps. First, a Pascal compiler translates the Pascal source code into intermediate code; then a code generator translates the intermediate code into 68000 machine code.

The linking process combines several machine language files to form the complete program. Apple supplies a number of files that must be linked with your program for it to take advantage of the Macintosh's built-

in software. These files contain external procedures that consist of a little extra code; chiefly, they connect the Pascal routines in the files declared in your USES section to the Macintosh's ROM routines.

Figure 3-2. Developing an Application



The linked program, however, is not yet a complete application. It has to be packaged at the resource level. You must combine all resource definitions for your application, including the linked program and the various definitions of windows, control, dialogs, and text. Currently, a resource compiler called "RMaker" does this for you.

In the current setup, each of these software tools, including the *compiler*, *code generator*, *linker*, and *resource compiler*, provides prompts asking for input files, options, and the names of output files. Fortunately, you can write command files called "exec" files, which automatically issue all commands and answer these prompts for you. You can invoke the entire process with a few keystrokes that start an exec file (discussed in Appendix B).

Pascal Pointers

Now let's look at *pointers*, a special part of Pascal that can cause confusion, yet that are very important to programming the Macintosh. Even if you know Pascal, this section merits attention because it explains the Apple Pascal implementation.

Pointers are important because they, in conjunction with the Macintosh's memory management system, allow dynamic variables. Such variables are needed to handle strings and essential parts of Macintosh's graphics interface to the user. We explore them at some length here because they are an aspect of Pascal that is important to understanding the Macintosh, yet may be poorly understood even by experienced Pascal programmers.

A Pascal *pointer* is a Pascal variable that references other Pascal variables (see Figure 3-3). All but the most minimal versions of Pascal have pointers, but they implement this feature differently. In Apple's Pascal compiler, pointers are implemented directly as memory addresses; that is, the *value* of the pointer is a memory *address*. Since addresses are stored as 32-bit integers by the 68000 processor, each pointer requires four bytes of storage.

Each pointer variable is *typed*. That is, it is defined to "point to" variables of a specific *data type*. The compiler rejects attempts to make a pointer reference data of the wrong type. However, Apple Pascal can very nicely convert data and pointers from one type to another. This data typing is performed by the compiler at compile time, not by your program as it runs. Thus, the type of a pointer requires no extra storage in your running program.

Realize that data typing is a tool to help protect your program from crashing as well as to help write better programs in shorter time. A caution:

any data typing should be done carefully and with good understanding of what you are doing to the machine. A debugging session might be necessary to achieve that understanding. In the next section, we describe such a debugging session.

As with any ordinary variable, pointers must be first declared in the VAR section of your program. Prefixing the name of any data type with a caret "^" creates the name of a new data type. In the original descriptions of Pascal, an upward arrow symbol was used. However, it is not possible to make such an arrow on the Macintosh, so the caret is used instead.

Variables of these new caret-prefixed types are called pointers because they "point" (that is, contain the address of) expressions of the original data types (see Figure 3-4). For example, starting with data type "INTEGER", the type "^INTEGER" is called "integer pointer" because variables of this type point only to integers. We can also assign a name to a pointer type in the TYPE section, then use this name as a known type in our VAR section. For example, in the TYPE section we could say

Figure 3-3. Pascal Pointers

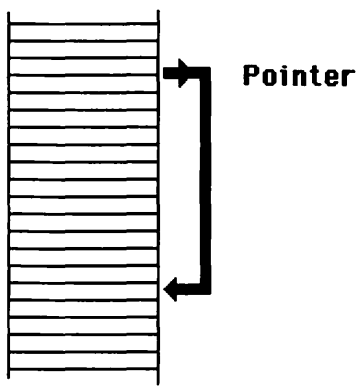
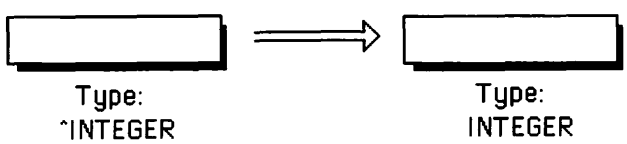


Figure 3-4. Pointer Data Typing



INTEGERPTR = ^INTEGER, then INTEGERPTR would be available as a valid data type.

Pointers can provide access information not readily available through other methods. For example, the information that controls the appearance of the video screen acts like a block of memory. We can direct a pointer to the various bytes of this memory, then turn on and off the individual bits that correspond to the dots (pixels) on the screen. In Chapter 4, you see an example program do this.

Pointers allow the user to access data in a form different than originally stored. For example, if you wish to examine the individual bits of an integer, you can set up a pointer of type:

```
PACKED ARRAY [0..15] of BOOLEAN
```

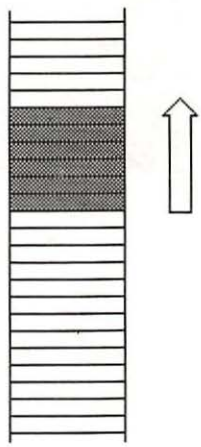
so that it points to your integer. Then you index this array to look at each bit. In Appendix C, we will do this to help examine and set certain file attributes.

Pointers and Dynamic Variables

Pointers also help manage memory by providing convenient methods of allocating and accessing *dynamic variables* (see Figure 3-5).

Dynamic variables move around in memory as they change in size. They are stored in areas of memory that are "relocatable".

Figure 3-5. Dynamic Variables



An example of a dynamic variable is a *region*. In Chapter 4 we study these in detail. Briefly, they store irregular shapes that are drawn on the screen (see Figure 3-6).

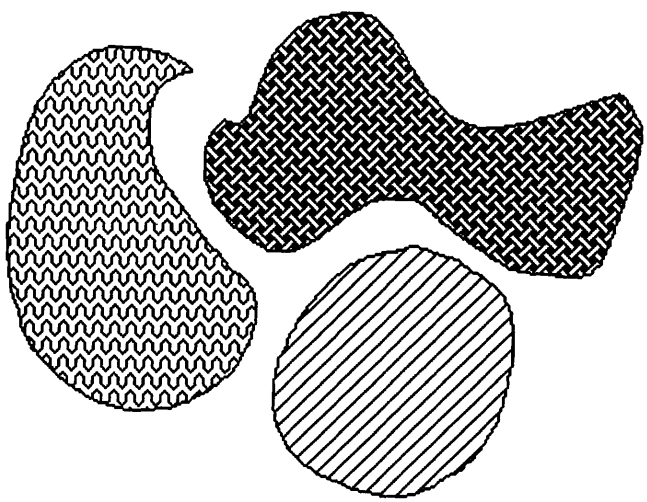
When you initialize a region (using a special function called "NewRgn"), it contains ten bytes of data. As you define and modify a region's shape, the amount of data usually grows, but occasionally shrinks. Each time the region needs more space than is immediately available, it moves to a memory area where there is room. Thus, it leapfrogs through memory, jumping over the storage space of other variables as it searches for more storage (see Figure 3-7).

Pointers can operate in various ways to form *pointer expressions* (formulas involving pointers). The compiler, however, prevents you from freely using such things as parentheses in these pointer expressions. This limits the ability to form pointer expressions.

The most fundamental pointer operator is the trailing caret "^". Use this whenever you want a pointer to store or retrieve data that it is pointing to. For example, if "theIPtr" is of type "^INTEGER", then "theIPtr" contains an *address*; whereas "theIPtr^" acts as an integer expression whose *value* is stored at that address.

When pointers manage dynamic variables, you often see *two* trailing carets after a pointer's name. In this case, the original pointer points to another pointer, which in turn points to the data (see Figure 3-8). For example, suppose that "the Rgn" is of type "RgnHandle" which is defined

Figure 3-6. Regions



as type "^RgnPtr", and that "RgnPtr" is defined as type "^Region". Then "theRgn" is a region *handle* containing the address of "theRgn^", a region *pointer* which in turn contains the address of "theRgn^^" which is where the actual data of the region is stored.

More generally, *handles* are regular "static" variables, providing the means for the programmer to maintain access to "dynamic" variables which the system continually and automatically moves around in memory.

Figure 3-7. Dynamic Variables Leapfrogging through Memory

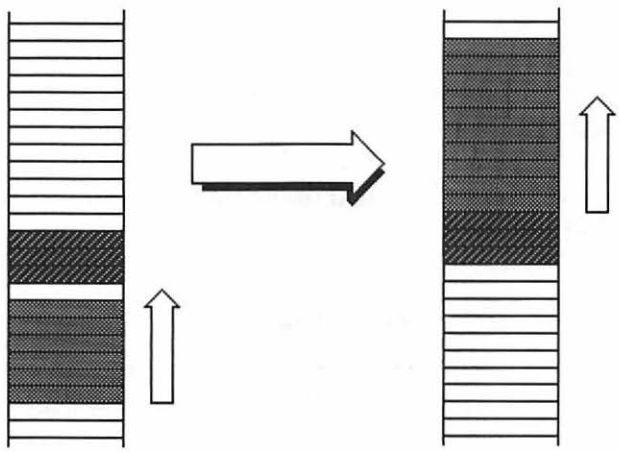
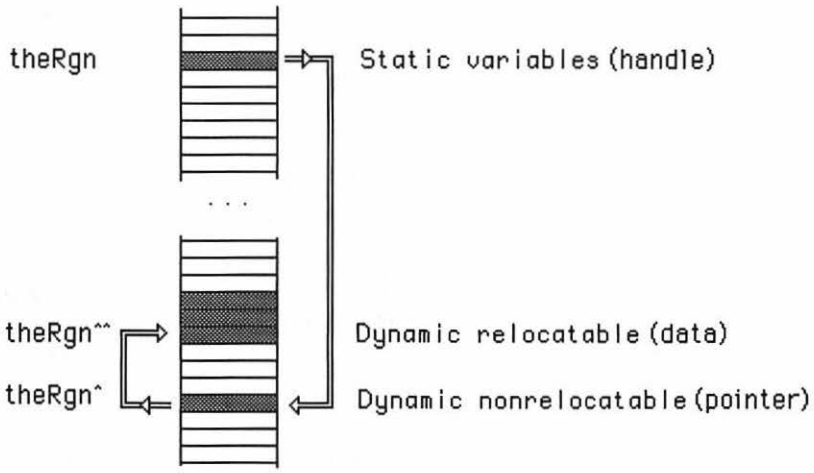


Figure 3-8. Handles, Master Pointers, and Dynamic Data



As the name implies, "static" variables do not move during the execution of the program. On the other hand, dynamic variables are used with data structures that change size, grow, or must be shifted in memory as the program runs.

The handle points to a pointer maintained by the Operating System in a special "nonrelocatable" area of memory called the *master pointer list*. Because the master pointer area is "nonrelocatable", these pointers don't move either.

The pointers in the master pointer area point to the actual data. When the Operating System moves data, it also updates the new location of the pointer on the master pointer list.

Thus, the handle can always access data by placing two carets after it. For example, if, as before, "theRgn" is a region handle (handle to a region), then "theRgn^" is in the master pointer area, and "theRgn^^" is the actual data of the region. The region data is a record structure whose fields can be accessed from this last expression. In particular, "theRgn^^.rgnSize" is the first field of the region's data. Such expressions are common in applications programs for the Macintosh.

Nonstandard Pointer Operators

So far, we have discussed standard pointer operators. Apple Pascal has other operators that make pointers into even more powerful programming tools.

Let's start with the "@" operator. When this operator is placed in front of a variable, it creates a pointer expression that points to that particular variable. This pointer expression can be assigned or otherwise passed to any pointer. Such a "typeless" pointer expression is said to be of type "NIL". For example, if "X" is of type "^INTEGER" and "Y" is any variable, then:

```
X := @Y;
```

makes "X" point to "Y". As a bonus, "X^" is an integer that reinterprets the first two bytes of Y as an integer.

The "ORD" and "POINTER" functions are also useful. The "ORD" function converts a pointer value into the corresponding numerical value. This numerical value is a long integer. For example, if Y is a variable, then:

```
ORD(@Y)
```

is the numerical value of the address where Y is currently stored.

The "POINTER" function converts a long integer (32-bit integer) into a pointer of type "NIL" that points to the memory location whose address is given by the long integer. For example, on the 128K Macintosh, the screen RAM is located at address $1A700. Thus,

```
POINTER($1A700)
```

is a pointer expression that points to the beginning of the screen memory. You should be warned that this is not where the screen is on other sizes of Macintosh. In Chapter 4, we explore a way to get this address for all Macintoshes.

The "ORD" and "POINTER" functions convert one type of pointer to another. For example, if "X" and "Z" are pointers of different types, then:

```
X := POINTER(ORD(Z));
```

assigns the address in Z to the pointer X.

Type Coercion

The last example can be more skillfully executed through the technique of *type coercion*. This method allows use of the name of a data type, just like a function, to convert one type of data to another. For example, if "X" is a pointer of type PTR, then:

```
X := PTR(Z);
```

is equivalent to the last example; that is, it also assigns the address in Z to pointer X. Another example is the expression:

```
LONGINT(X)
```

which is the numerical value of the address contained in the pointer "X". This can substitute for the POINTER function.

Be aware that the compiler doesn't like to use type coercion when two types of data require different amounts of storage. This can be overcome by changing types at the pointer level, since all pointers require the same amount of storage — four bytes (32 bits). For example, if "numPtr" is defined in the TYPE section as:

```
numPtr = ^numArray,
```

where "numArray" is also defined in the TYPE section as:

```
numArray = ARRAY [1..1000] OF INTEGER
```

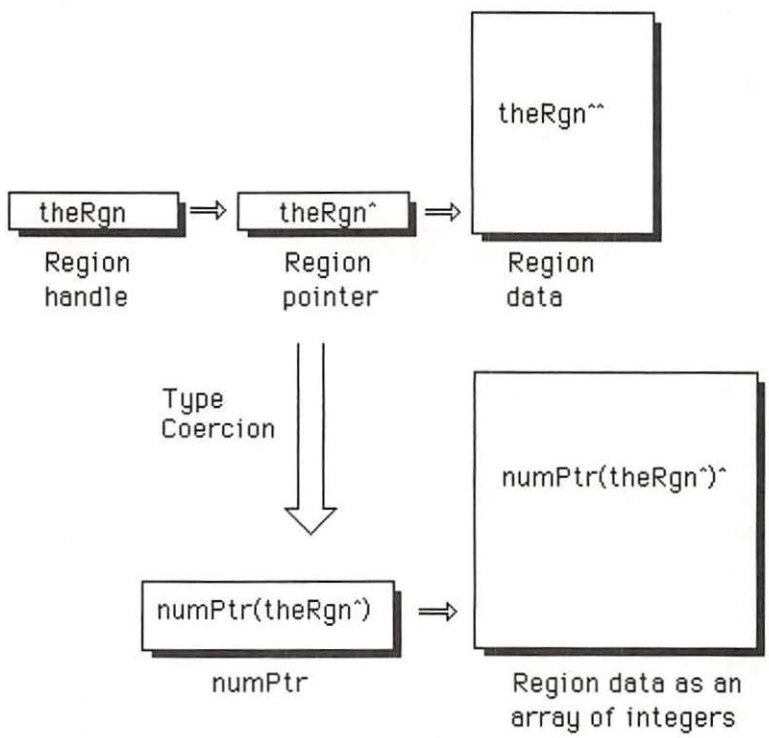
then

```
numPtr(theRgn^)^
```

is an array of integers that contains the region data (see Figure 3-9). In Chapter 4, we use this technique to explore how regions work.

Also be careful about the values you get when using type coercion: the results depend upon how data is stored. For example, if X is an integer, then the expression Ptr(@X) is a byte pointer that points to where X is located. However, the value of the byte there, as given by the expression Ptr(@X)^, is equal to the upper byte of X (not the lower byte as with some

Figure 3-9. Converting Data to Integer Format



processors). The upper byte value of X hardly ever equals X itself, whereas the lower byte equals X if X is small enough.

Using a Debugger

Now that we've seen how pointers work, let's see how a debugger can chase them through memory.

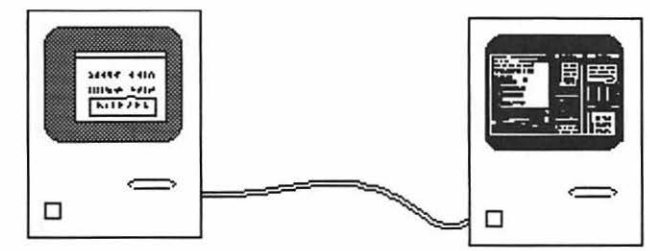
Debuggers are essential to understanding your program, especially when it's not doing what you wanted it to. Such knowledge can save hours. With a debugger you can follow the execution of your program in the machine step by step, examining the CPU registers and memory at will.

We describe the process of running an example program on one Macintosh while using a second Macintosh to display debugging information (see Figure 3-10). Even if you are using only one Macintosh as a debugger, you should have no trouble following the discussion.

The two-Macintosh method has the advantage of two display screens: one to show program output, one to show all debugging information. Since applications programs use the screen so completely, it is very helpful to send debugging information to a second screen. Furthermore, the debugging program used here works very effectively, filling the screen of the second Macintosh with windows containing information such as the state of the CPU registers, the stack, and a section of code of the program concurrently being executed. It also allows other windows to open showing other areas of code or areas of data.

Currently, debugging programs are available from Apple as part of the Macintosh development system. However, programs from other companies will no doubt appear.

Figure 3-10. Mac-to-Mac Debugging



Hardware Setup

A cable connects the two machines via their printer serial ports. Figure 3-11 shows the pin assignments for this cable. The ports use an RS-422 standard for serial transmission. This is a little different from the usual RS-232 standard but is compatible using slightly different cabling.

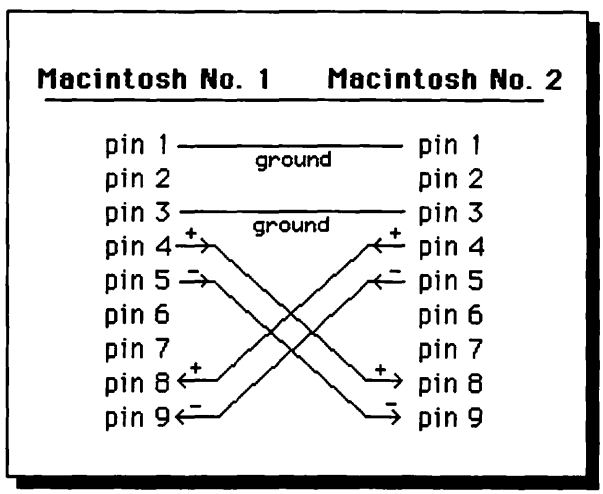
The connector on each Mac printer port is a DB-9; that is, a D-shaped connector with nine pins. As shown, not all the pins are used.

Pins 1 and 3 are ground. Pin 2 (not used) is +5 volts. Pin 4 is data out plus, and pin 5 is data out minus. Pin 8 is data in plus, and pin 9 is data in minus. Two lines are required for each direction to conform with the RS-422 standard. Instead of comparing signals to a common ground, they are compared against each other. This reduces noise and allows longer transmission lines.

Required Software

To make this system work, you must execute a program (currently called "MacNub") on the first Macintosh (the one with your program). The "MacNub" program loads certain interrupt vectors and interrupt service routines into the Macintosh. These routines take over the basic functioning of the Macintosh, allowing debugging information to be sent out as the machine executes subsequent programs.

Figure 3-11. Pin Assignments



You must also run a program (currently called "DB" or "MacDB") on the second Macintosh (the debugging Macintosh). This program controls the display of the debugging information on the second Macintosh. Figure 3-12 shows a typical display.

The Example Program

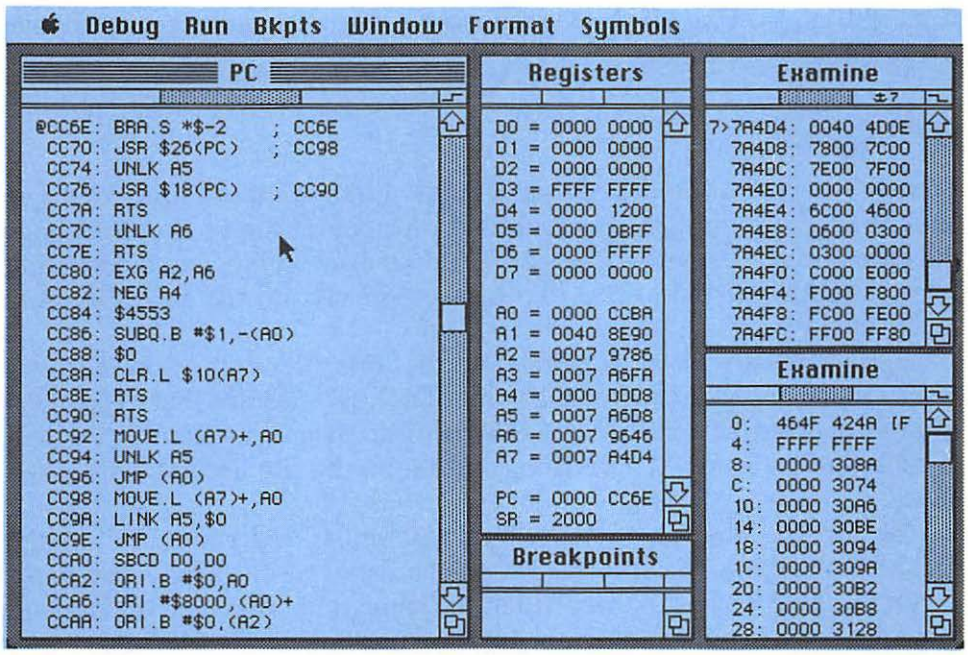
Our example program for this debugging session initializes a region and then goes into an endless loop. The program signs on and then just waits for you to hit the interrupt button (to debug) or reset button (to restart the machine).

As we debug this program, we carefully check some examples of the pointers that we presented in the previous section. We explain this checking procedure after discussing the program.

Here is the program:

```
PROGRAM Endless;
   { $R-}{$X-}
```

Figure 3-12. Debugging Screen



```
USES
   {$U obj/Memtypes    }Memtypes,
   {$U obj/QuickDraw   }QuickDraw,
   {$U obj/OSIntf      }OSIntf,
   {$U obj/ToolIntf    }ToolIntf;

VAR
   theRgn: RgnHandle;

BEGIN
   theRgn := NewRgn;
   WHILE TRUE DO;
END.
```

The USES section, as in our earlier program, simply declares the external library files that can be used.

The VAR section declares the variable "theRgn" to be of type "RgnHandle".

In the main part of the program, the first line invokes the function "NewRgn". In Chapter 4, we explain it in detail. For now, understand that this function allots room for and initializes a region, then returns a handle to the region data. In this program, we assign this handle value to the variable "theRgn".

The second line of the main program is an infinite WHILE loop. The program continually executes this loop until you intervene by hitting the reset button, turning off the Macintosh, or using the debugger to change the program counter.

Debugging

Now let's describe the debugging process. First, compile, link, package, and transfer your program as described previously. Don't forget to change the name from "Trivial" to "Endless" in the exec and resource definition files. If you are using a Lisa, the Lisa editor can do the searching and replacement for you.

Let's say we ran the exec file, and the finished application is stored on a Macintosh disk under the file name "Endless". Assume that "MacNub" is on the same disk and that this disk is inserted into a Macintosh which is cabled to a second Macintosh according to the pin assignments given above.

Now run "MacNub" on the first Macintosh and run "DB" on the second (debugging) Macintosh. The exact order of running these programs doesn't matter. They can even run at the same time. However, understand that "MacNub" can run only once without restarting the Macintosh.

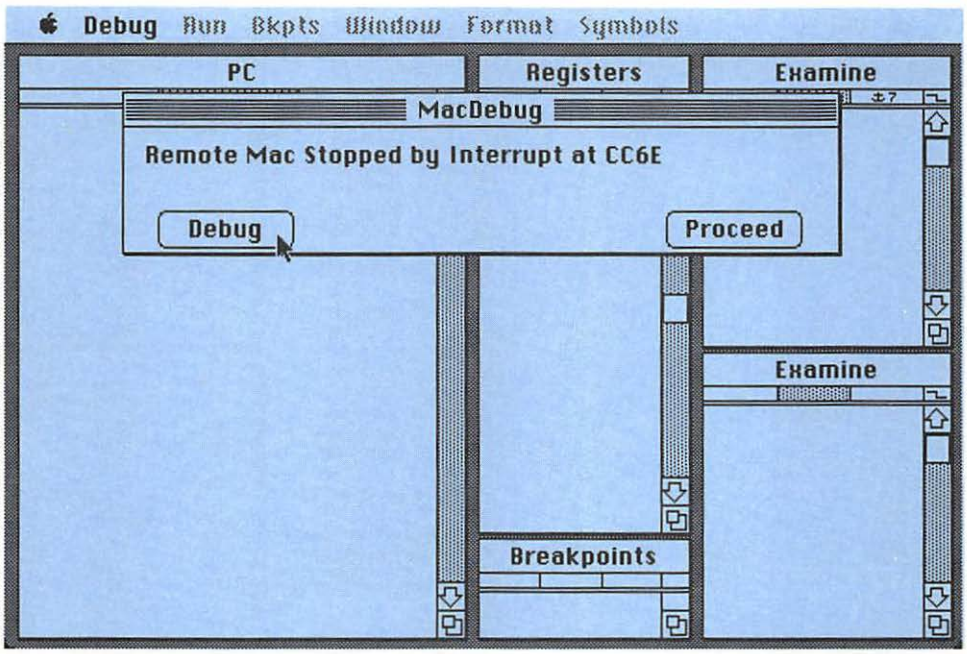
After "MacNub" is run on the first Macintosh, start up our "Endless" program. The second Macintosh should now tell you, with a dialog box, that it is waiting for an interrupt.

When the disk stops on both Macintoshes, hit the interrupt key on the first Macintosh. This key is just behind the reset key. The second Macintosh's screen should now tell you where the first Macintosh stopped (see Figure 3-13). This should be on the WHILE loop.

Now use the mouse to select the DEBUG button on the screen of the second Macintosh. You now see the debug windows fill with information. You can scroll the window to view the entire machine code for the program in the "PC" (Program Counter) window (see Figure 3-14). The location of the program counter itself is indicated by an "@" in front of the address that it points to. This is the address of the next instruction to be executed at the time that the interrupt key is hit. In our example this is a "BRA.S" (branch short) instruction to the same address. This is how the endless WHILE loop is translated into machine language.

Now let's check out the region handle. Just before the "BRA.S" instruction there should be a "MOVE" instruction, and before that should be a "NewRgn" command. The "NewRgn" allocates space for and ini-

Figure 3-13. Got an Interrupt

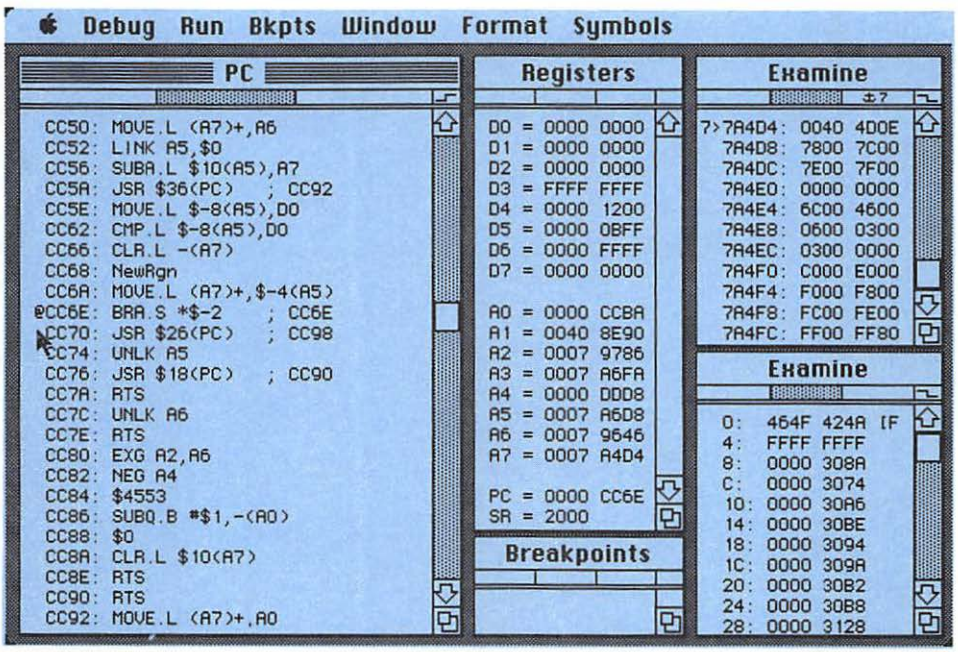


tializes a region. It returns a handle to this data. This return value can be found on the *stack*. Register A7 is the *stack pointer*. The "MOVE" instruction moves the region handle from the top of the stack (using register A7) to "theRgn". Here, the address of "theRgn" is given by (A5) – 4; that is, four less than the address stored in register A5.

Register A5 is a key to locating data used by Macintosh programs. When the Operating System starts an application, it loads A5 with the address of the beginning of an area in the stack called the application parameter area (see Figure 3-15). This area contains variables that are outside of your Pascal program but are shared by the application and the Operating System.

Just below the application parameter area are the program's global variables. These are also in the stack and, like all stack variables, are laid down in memory at successively lower addresses as they are placed on the stack. As a result, they appear in memory in the reverse order of that in which they are declared (see Figure 3-16). When your program references global variables, you will see addresses that use A5 with a negative offset.

Figure 3-14. Displaying the Program



Our example has just one global variable, "theRgn" (see Figure 3-17). Since it requires four bytes of storage, it is located at four bytes less than the address in A5. This is indicated by the address reference code "$-4(A5)" in the "MOVE" instruction.

Global variables declared in external library UNITs are placed after your global variables. They are referenced with negative offsets with larger

Figure 3-15. Application Parameter Area

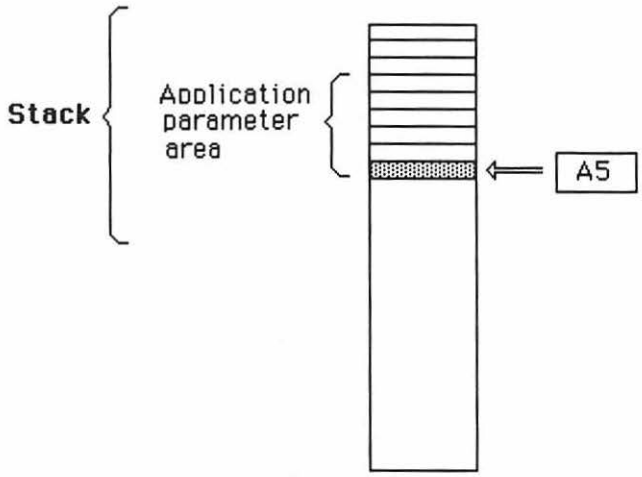
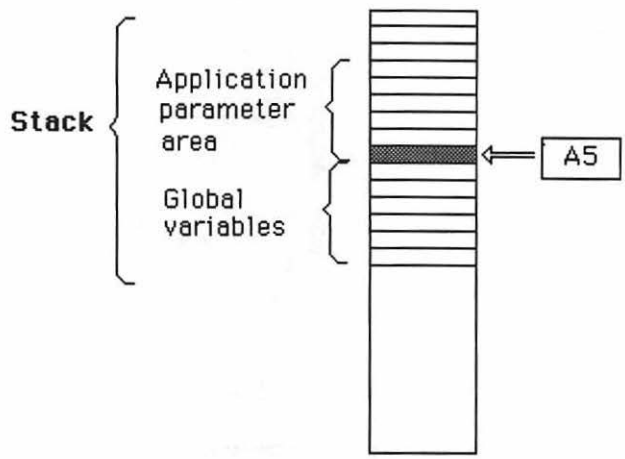


Figure 3-16. Global Variables



magnitudes than those for your own global variables. These parameters are used extensively throughout the book.

Now that we have located our region handle "theRgn", let's follow it to the data (see Figure 3-18). The exact addresses vary from system to system, but the method is the same. On our Macintosh, we find the address $7A6D8 in A5. Subtracting 4 gives the address of "theRgn" as equal to $7A6D4. We create a window displaying the memory locations around

Figure 3-17. Referencing Our Global Variable

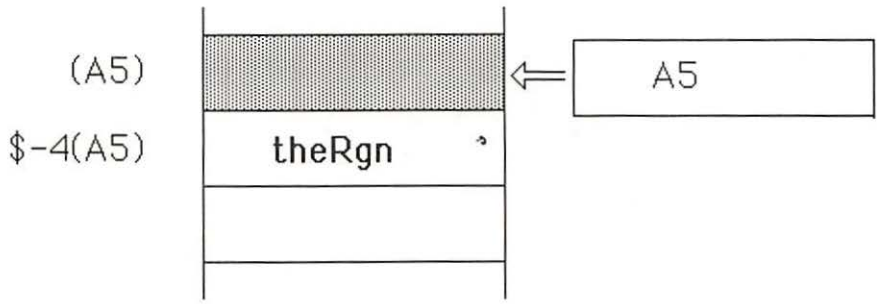


Figure 3-18. Chasing Our Region Handle

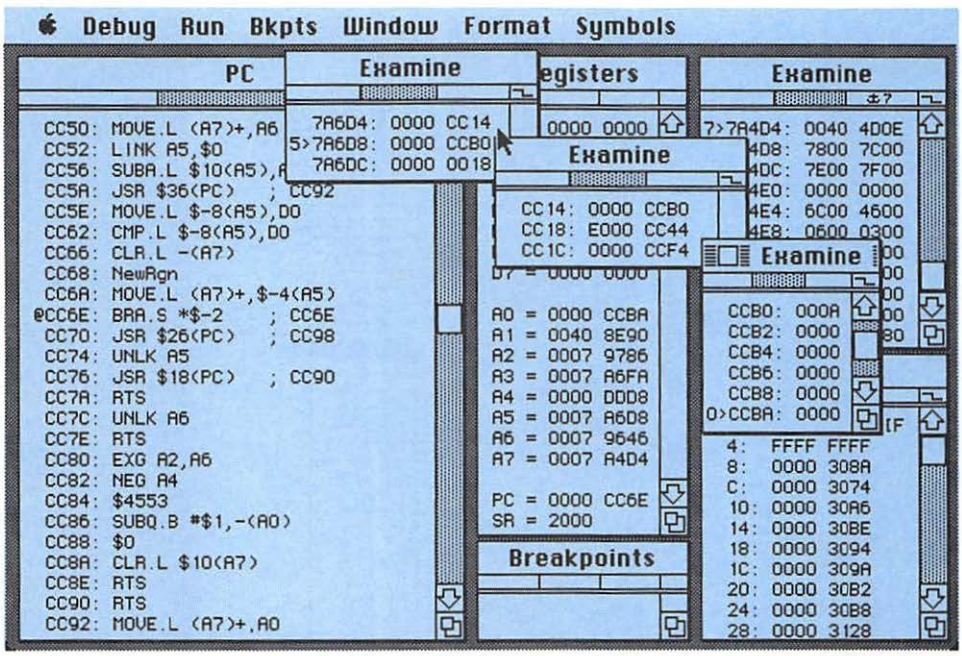


Figure 3-19. The Heap and the Stack

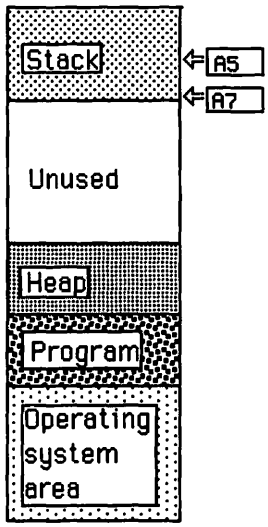
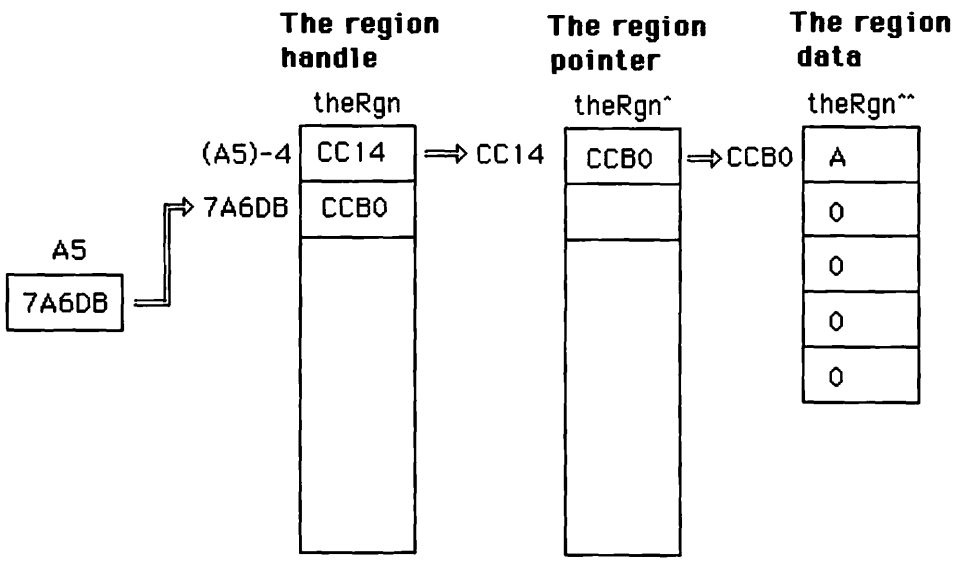


Figure 3-20. Debugging Session



this address. In this window we see that the address $CC14 is stored in "theRgn". This is the address of the region pointer "theRgn^". We create another window displaying the locations around the pointer. Looking into this window, we see that the address stored in the pointer is $CCA8. This is the address of the region data. We create one more window to show this data. The first word contains "000A", which is the length (10 bytes) of the region data area. Initially, the remaining eight bytes contain zeros, as shown.

Notice that the addresses of the region pointer and region data are much lower than the address in A5 and the address of the region handle. That's because the region pointer and the region data are in an area called the *heap* (see Figure 3-19). In contrast to the stack, which starts high in memory and grows downward, the heap starts low in memory and grows upward. All dynamic variables are maintained in the heap.

Figure 3-20 shows the relationships in the debugging session as a whole.

Use this debugging system and the lessons learned here to explore other example programs. See how efficiently the compiler turns our Pascal source code into 68000 assembly language. Use the "Run" and "Bkpts" menus to step slowly through these programs to thoroughly understand how the Macintosh works.

Summary

This chapter has discussed three important aspects of programming the Macintosh: the programming environment, special features of the programming language, and the debugging environment.

The exact programming steps and language may differ in the system that you use, but this chapter should give you a good start towards working in any program development environment for the Macintosh.

4

QuickDraw

This chapter covers the following new concepts:

- **QuickDraw**
- **QuickDraw Initialization and Default Variables**
- **External Units**
- **GrafPorts**
- **Memory-Mapped Video**
- **Pixels, Patterns, and Cursors**
- **BitImages and BitMaps**
- **Coordinate Systems**
- **Points, Rectangles, and Regions**
- **Visibility and Clipping**
- **Local Coordinates**
- **GrafPort Moving Routines**

This chapter introduces QuickDraw, a collection of routines and data that draw pictures on the Macintosh's video screen. We present several short example programs that illustrate its basic structure and features.

QuickDraw is essential to Macintosh's built-in software: it produces the graphics that are the primary means of conveying information from the Macintosh to the user.

QuickDraw routines are carefully optimized for speed and power, providing a rich set of drawing primitives representing years of development. By taking full advantage of these routines, you can save considerable time, effort, and overhead to your program, and most likely get better results.

QuickDraw routines govern a variety of shapes, from points to polygons to irregularly shaped objects called regions. Each shape responds to such actions as framing, filling, and erasing. These shapes are very useful, as seen in our example programs.

The Macintosh's Operating System and its applications call upon the same set of QuickDraw routines. For example, when you boot up a disk and see the desktop with icons and windows indicating disks, folders, data files, and program, you are looking at images created by QuickDraw. Applications such as MacWrite and MacPaint use QuickDraw to draw characters, borders, patterns, and shapes such as rectangles, ovals, and irregular shapes. Even programming languages such as Macintosh Pascal and BASIC call upon QuickDraw routines to display all program listings and output. In short, everything that normally appears on the Macintosh's screen is produced by QuickDraw.

Within the Operating System, Toolbox managers such as the Window Manager and the Control Manager (see Chapters 6 and 7), the Dialog Manager (see Chapter 8), the Menu Manager (see Chapter 9), and Text Edit (see Chapter 10) call QuickDraw directly to draw the various objects that they manage. Other Toolbox managers, such as Dialog Manager, call upon the Window Manager and Control Manager as well as QuickDraw to draw objects on the screen.

This chapter begins by discussing the proper initialization of QuickDraw. An example program shows you minimal steps needed to do any drawing on the Macintosh's screen.

Next is a discussion of *patterns* and *cursors*. We explain what they are and how they are used. An example program shows how to define your own patterns and cursor shapes and how to make them appear on the screen.

A discussion of memory-mapped video explains the basic hardware of the screen display, leading to a discussion of *pixels*, *points*, and *coordinate systems*. An example "Mouse points" program shows how to map any position of the mouse directly to the screen. This is the only example that does not use QuickDraw to draw on the screen. It demonstrates how the screen works, and it also illustrates the overhead saved by using QuickDraw.

Rectangles and then *regions* are discussed next. These fundamendal constructs form the basis of much that QuickDraw can do. *Rectangles*

define and draw such things as windows and menus. *Regions* draw regular and irregular shapes, such as the parts of a window (grow box, drag area, and goAway box) and controls (scroll bars, buttons, and check boxes). *Regions* are also used in conjunction with mouse tracking; that is, finding the mouse and responding with proper highlighting. *Regions* are also used by the Window Manager to handle overlapping windows. Example programs show how to define and draw rectangles and how the internal structure of regions works.

The chapter finishes with a discussion of *grafPorts*. A *grafPort* is a full set of parameters that defines the drawing environment. Understanding *grafPorts* is essential to understanding QuickDraw. This is particularly necessary if you use a debugger on QuickDraw programs. An example program shows how to use grafPorts to good advantage, switching rapidly among several grafPorts to illustrate how multiple windows can be implemented in a concurrent fashion.

Not all of QuickDraw is covered here — just enough to explain the fundamentals and show how to write programs that make useful drawings. A complete discussion of every QuickDraw feature would require an entire book. In subsequent chapters, we add more QuickDraw routines to our programming arsenal.

After reading this chapter, you should understand and be able to properly initialize the Macintosh's drawing environment and draw shapes on the screen.

Initialization

Every applications program for the Macintosh requires an *initialization* process. This process ensures that certain variables contain proper values and certain subsystems are set up in the proper configuration. We are concerned here with QuickDraw and its initialization, including that of its working variables and the hardware that controls the display screen.

We begin with an example program that demonstrates how to initialize QuickDraw. When we examine this program we will see the specific things that need to be initialized and investigate their general structure, but understanding much of their detailed structure must wait until you have more background, which is provided later in this chapter.

The Initial Example Program

Our first example program performs a series of steps that initialize QuickDraw and the video screen. The program stops at a couple of key steps and waits for you to press the mouse button before proceeding to

the next step. This allows you to carefully examine what is happening on the screen. Otherwise, the application would immediately wipe out the results and begin returning to the Operating System.

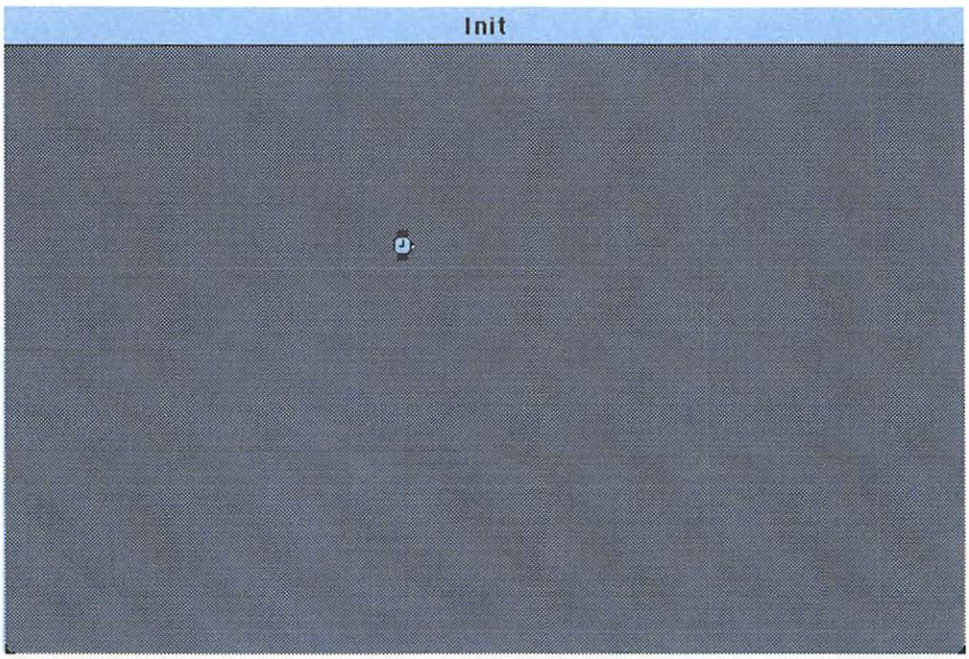
The program first displays a gray desktop with a white band where the menu bar is usually located along the top of the screen (see Figure 4-1). A title, "Init", appears in this white area. At this point, the cursor is the familiar "waiting watch" (see Figure 4-2). This is the condition of the screen before the applications program begins.

Stopping here is very helpful if you want to use a debugger to see how QuickDraw is initialized. You would simply hit the interrupt switch at this point and the debugger takes over (if properly set up). You could then use the debugger to step through the ROM routines.

If you now click the mouse button, the cursor turns into the standard arrow (see Figure 4-3). At this point, QuickDraw's default variables, a grafPort, and the cursor are all properly initialized. The first two steps do not change the appearance of the screen. We do not pause until after the third step, which does change the screen.

The appearance of the arrow cursor normally tells the user that everything required has been loaded and initialized, and to proceed. In this

Figure 4-1. First Screen of the First Program

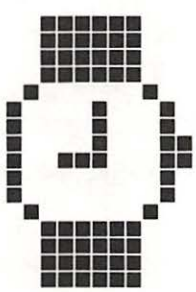


program, to proceed means one more click, which causes the program to terminate.

Here is the program:

```
PROGRAM Init;
   { $R-}{$X-}

   USES
      {$U obj/Memtypes    }Memtypes,
      {$U obj/QuickDraw   }QuickDraw,
      {$U obj/OSIntf      }OSIntf,
      {$U obj/ToolIntf    }ToolIntf;

   PROCEDURE ClickButton;
      BEGIN
         While Button DO;
         While NOT Button DO;
         While Button DO;
      END;

   BEGIN  {main program}
      {Wait before doing anything}
      ClickButton;

      {Initialize QuickDraw default variables}
      InitGraf(@thePort);

      {Allocate nonrelocatable space on heap for grafPort}
      NEW(thePort);

      {Initialize grafPort}
      OpenPort(thePort);
```

Figure 4-2. The Waiting Watch



```
    {Initialize the cursor and wait}
    InitCursor;
    ClickButton;
END.
```

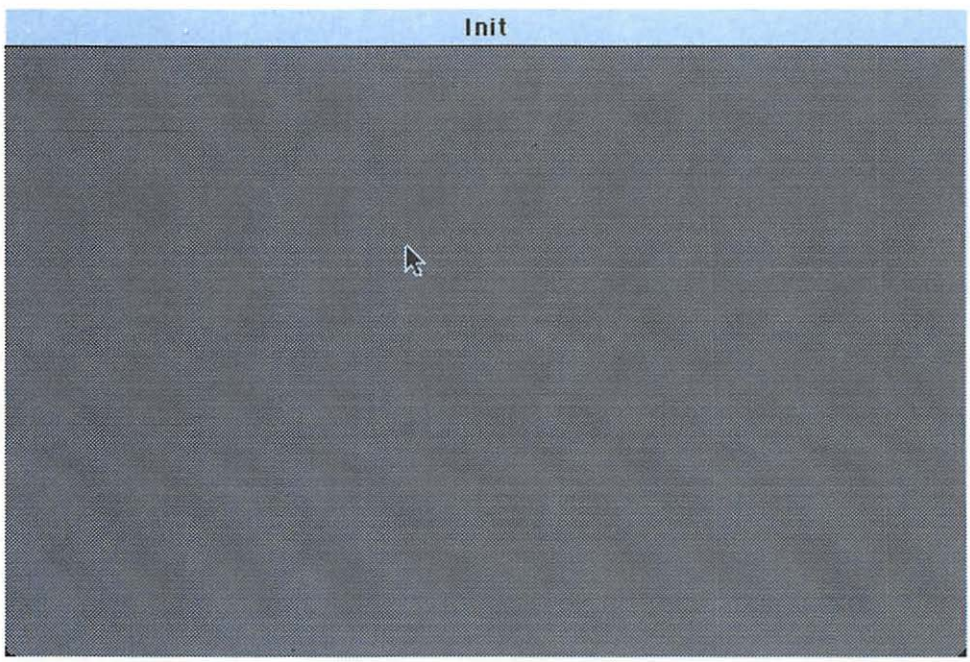
External Files

We begin with the USES section, since we studied the *program statement* on the first line and the compiler commands on the second line in Chapter 3. Chapter 3 also mentioned the external files listed in the USES section. We study these in detail here.

The USES section of this program allows the program to take advantage of the large number of Pascal declarations developed by Apple to interface to the Macintosh's ROM.

These declarations are stored in several compiled external files that are referenced in the USES section. Each external file can contain one or more UNITs. A UNIT is a special kind of Apple Pascal module that interfaces data and procedures to a Pascal program. The data and procedures may be contained within the UNITS or may reside elsewhere, such as in the ROM.

Figure 4-3. Second Screen of the First Program



The external library files are distributed to developers as text "source" files and in compiled form.

Each UNIT begins with a UNIT statement that names the UNIT. The UNIT itself has two parts; an INTERFACE section and an IMPLEMENTATION section. The INTERFACE section declares those data types, data variables, and procedures that should be *public*. The IMPLEMENTATION section contains the types, variables, and procedures that should be *private*. The public entities are available to any program that USES the UNIT; the private entities are local to the UNIT.

The compiler command $U followed by a file name tells the compiler to search a particular file for any UNITs referred to subsequently during any USES section of the program. In our program, each UNIT is in a separate file.

The first UNIT in our USES section is called "Memtypes" and is contained in the file "obj/Memtypes". In our program, this *file name* appears *within* the comment delimiters immediately after the first $U command. On the same line, but after the comment, comes the *UNIT* name "Memtypes".

The "Memtypes" UNIT declares certain data types used by all parts of the Macintosh's ROM, including QuickDraw. The definitions of types such as "SignedByte", "Byte", "Ptr", and "Handle" reside in this library file. These declarations are in the INTERFACE section. The IMPLEMENTATION section is empty.

The second UNIT in our USES section is called "QuickDraw", located in file "obj/QuickDraw". This UNIT contains a CONST section, which defines a number of public constants that identify various pen modes and color information.

The "QuickDraw" UNIT also contains a TYPE section, which defines QuickDraw data types: "QDByte", "QDPtr", "Pattern", "Style", "Point", "Rect", "grafPort", and others. We encounter many of these data types throughout this book.

The QuickDraw UNIT also has a VAR section, containing QuickDraw's public default variables. We study these as part of the initialization of QuickDraw in the main part of this program. Following the VAR section are declarations of all QuickDraw procedures and functions. The bodies of these procedures and functions generally consist of special "inline" machine-language calls to ROM routines (see Chapter 2).

The IMPLEMENTATION section of the "QuickDraw" UNIT consists of a compiler command to include the file "obj/Quickdraw2", where the contents of the implementation section are actually located. Some private QuickDraw variables are declared here, which we discuss later.

Two other UNITS, "OSIntf" and "ToolIntf", are also invoked in our USES section. These allow us access to the "Button" function that appears in the routine "ClickButton", which is used to pause for button clicks in our program. Many of the data types, data variables, and procedures in these UNITS are discussed in later chapters. Each example program requires all of the UNITS that appear in the USES section of this program.

The ClickButton Procedure

Our program has one procedure called "ClickButton", which waits for a click of the mouse button. It consists of three WHILE loops that involve the Toolbox's "Button" function. In each case, the DO part of the loop is empty.

The "Button" function is not part of QuickDraw. We have borrowed it because it provides the easiest and most elementary way to allow the user to control the progress of the program. In Chapter 5, we see how *events* provide better control of a program.

The "Button" function returns a Boolean value that is true if the mouse button is down, false if it is up.

Let's see how the "ClickButton" routine works. When this routine is called, the button can be up or down. If the button is down, then the first WHILE loop is executed until the button is released. The second WHILE loop executes as long as the button remains up. If we now press the button down, the second loop terminates and the third loop begins, continuing until the button is released.

If the button is up, the first WHILE loop never fully executes, and the second WHILE loop executes so long as the button remains up. If we now press the button down, the second loop terminates and the third loop executes until the button is released.

In either case, the routine terminates when the button is pressed, then released.

The Main Program

The main part of the first example program begins by calling our "ClickButton" routine. This stops the program before it does anything. Again, this is useful when using a debugger to see exactly how QuickDraw performs its own initialization.

Initializing Default Variables. Following "ClickButton" is a call to "InitGraf". This call is required by most programs that run on the Macintosh. It initializes all QuickDraw default variables.

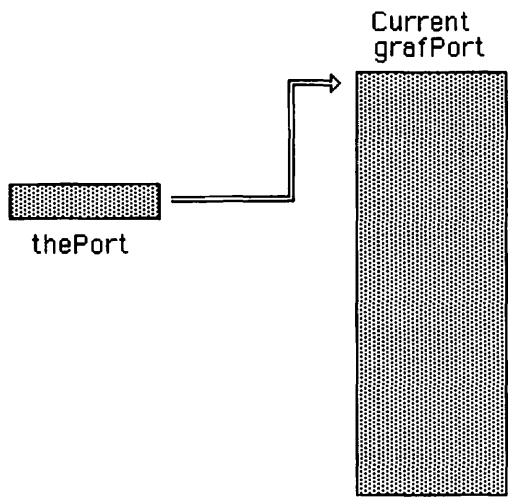
The QuickDraw "InitGraf" routine expects one parameter, namely a pointer to the variable "thePort", the first of QuickDraw's public default variables. As discussed in Chapter 3, placing an "@" before a variable creates a pointer to that variable.

In this case, "thePort" is a pointer to the current *grafPort*, a data structure containing all current drawing parameters such as pen size and pattern, text size and style, and the active drawing area (see Figure 4-4). As the chapter goes on, we explore this concept of *grafPort*, culminating in an example program where we see how to make "thePort" point to a series of grafPorts to achieve interesting special effects. Here, however, the initialization routine "InitGraf" uses "thePort" mainly as a place holder. In fact, at this point in the program, there is no grafPort.

Default variables initialized by "InitGraf" are declared in the external UNIT "QuickDraw". For graphics programs to operate properly, you must include "QuickDraw" in the USES section of your program, as we have done here. Then "thePort" and other default variables automatically become global variables in your program. The "InitGraf" routine uses the position of "thePort" to locate other default variables; thus, default variables must follow "thePort" in the exact order specified by the "QuickDraw" UNIT.

This particular scheme allows you to use a built-in routine (namely, "InitGraf") to initialize variables that belong to your program. Since your program may be loaded into different places in memory and these variables

Figure 4-4. ThePort and the GrafPort



may be placed in different positions within your program, you must tell this routine where to find these variables. There are several ways to do this. However, "InitGraf" expects you to use the location of the first default variable to specify where all these variables are located. Thus you must use this method.

If you examined the "QuickDraw" file, you would see the public default variables declared in the VAR section of the "QuickDraw" UNIT as follows:

```
thePort:     GrafPtr;
white:       Pattern;
black:       Pattern;
gray:        Pattern;
ltGray:      Pattern;
dkGray:      Pattern;
arrow:       Cursor;
screenBits:  BitMap;
randSeed:    Longint;
```

The first variable is "thePort", which we just described. Note that "thePort" is not the actual grafPort, merely a pointer to it. Calling "InitGraf" initializes "thePort" to NIL, indicating that there is *no* current grafPort without further action. Be patient; we will get to a real grafPort soon.

The next five public default variables are "patterns" permanently available for filling or "painting" objects on the screen. Whenever you need a white, black, gray, light gray, or dark gray pattern to fill a rectangle, oval, or irregular shape, you simply pass the name of one of these variables.

The next default variable, "arrow", contains data for drawing the familiar arrow cursor. In a later example program, we see how to define our cursor shape.

Next comes a "BitMap" called "screenBits", which describes the Macintosh screen. This default variable is useful to define rectangles that cover the entire screen. In later chapters, they are used to set the limits for dragging "windows" around the screen.

The variable "screenBits" also serves as a template for setting up certain fields of the grafPort. We study BitMaps in detail when we study grafPorts.

The last public default variable, "randSeed", is a seed for a random number generator, which, surprisingly, is part of QuickDraw.

Now that we have quickly surveyed QuickDraw's public default variables, let's list the private default variables, also initialized by "Initgraf":

```
wideOpen:       RgnHandle;
wideMaster:     RgnPtr;
wideData:       Region;
rgnBuff:        QDHandle;
rgnIndex:       Integer;
rgnMax:         Integer;
playPic:        PicHandle;
playIndex:      Integer;
thePoly:        PolyHandle;
polyMax:        Integer;
patAlign:       Point;
fontAdj:        LontInt;
fontPtr:        FMOutPtr;
fontData:       FMOutRec;
```

The data types range from regions and associated pointers to a complex record structure called FMOutRec. They are defined in the IMPLEMENTATION section of QuickDraw, contained in the file "QuickDraw2". They are only for internal use by QuickDraw's ROM routines, so we do not discuss them in detail.

Initializing a GrafPort. Now let's talk about *grafPorts*. To really understand what a *grafPort* is, you must know something about how graphics are programmed on a computer.

Basically, a graphics program generates a sequence of graphics *commands* that produce the picture on the screen or other graphics device, such as a plotter or printer. These graphics commands generally fall into two classes: *drawing commands* and *attribute-setting commands*. The *drawing commands* actually produce immediate visible results, such as lines or rectangles that appear on the screen, whereas the *attribute-setting commands* set parameters (attributes), such as line width, that affect how subsequent *drawing commands* are executed.

As *attribute-setting commands* are executed, their effects accumulate. For example, if you set the text size and then the text style, both remain in effect until you change them.

If we didn't use attributes in this way, we would have to pass a large number of parameters with each drawing command. Since many of these parameters remain the same for long periods, this would be considerably less efficient than using separate attribute commands.

A *grafPort* provides storage for the drawing attributes while they are in effect. Later, when we study grafPorts in detail, we will see what these attributes are for the Macintosh.

For now, understand that a *grafPort* is a Pascal *record structure*. That is, it is identified by a single name and yet contains a number of *fields* holding variables which can be any specified data type. For the *grafPort*, there are 24 individual fields ranging from integers to handles to complex data structures such as *patterns*, which we will study later in this chapter.

Storing all attributes in one structure allows one to quickly switch from one set of attributes to another. This is useful for producing dynamic, interactive displays in which the parts of one or more pictures are maintained at once. The "Ports" program at the end of the chapter illustrates this.

When you begin to draw, each attribute has a default setting. For example, lines are one unit wide and text has a plain style. Unfortunately, the programmer has to actually execute a command to set up these default attribute values.

Now let's return to our example program to see how the next couple of statements in the main program initialize a grafPort.

The first of the two statements invokes Pascal's "NEW" procedure to allocate space for the grafPort. The second statement calls QuickDraw's "OpenPort" to initialize the various fields of the grafPort.

The "NEW" command expects a single parameter — a pointer to the particular type of data structure that we wish to allocate space for. The "NEW" command allocates an area in memory of the correct size and places its address in the pointer so that it now points to the newly allocated structure.

In our program, we pass "thePort", which is of type "grafPtr", defined in the TYPE section of the UNIT QuickDraw as:

```
grafPtr = ^grafPort;
```

where "grafPort" is the Pascal record structure defined by:

```
grafPort = RECORD
             device:     INTEGER;
             portBits:   BitMap;
             portRect:   Rect;
             visRgn:     RgnHandle;
             clipRgn:    RgnHandle;
             bkPat:      Pattern;
             fillPat:    Pattern;
             pnLoc:      Point;
```

```
    pnSize:      Point;
    pnMode:      INTEGER;
    pnPat:       Pattern;
    pnVis:       INTEGER;
    txFont:      INTEGER;
    txFace:      Style;
    txMode:      INTEGER;
    txSize:      INTEGER;
    spExtra:     INTEGER;
    fgColor:     LongInt;
    colrBit:     INTEGER;
    patStretch:  INTEGER;
    picSave:     QDHandle;
    rgnSave:     QDHandle;
    polySave:    QDHandle;
    grafProcs:   QDProcsPtr;
END;
```

So this a *grafPort*. The entire structure sits in one continuous stretch of memory. Within this structure are 24 fields, each with its own distinct structure and use. Later, we will go through a number of these fields, but we need some more background before that is possible. Meanwhile, let's continue with this program.

The "OpenPort" routine expects one parameter of type "grafPtr", which points to a grafPort that has been allocated but not yet initialized. The routine "OpenPort" initializes the various fields of this structure. We will indicate their "initial" values as we study them individually.

Initializing the Cursor. The last initialization step of our program calls "InitCursor" to turn the cursor into the familiar arrow cursor that is stored as one of our default variables.

The "InitCursor" routine expects no parameters. It simply sets the cursor equal to the standard arrow shape. In our next example program, we see how to define our own cursor shape.

The last step of our program calls "ClickButton" again to pause the program before we allow it to terminate.

You now know the fundamentals of initializing the variables in QuickDraw routines. You have a list of all QuickDraw default variables and a list of all fields of a grafPort. We study some of these in the remaining part of the chapter.

Cursors and Patterns

Let's look at cursors and patterns, some of the most fundamental default variables, and how they are defined. An example program demonstrates

how to define and draw your own patterns and cursors. This gives you a good start in learning about default variables and the fields of a grafPort. At the same time, you see interesting results on the screen.

Defining Patterns

The Macintosh does not have color: *patterns* distinguish different areas on the screen. They fill or "paint" shapes such as rectangles, ovals, and polygons. In this section, we fill the entire screen with a crosshatched pattern.

The Macintosh's video screen, like most modern graphics screens, can be thought of as a two-dimensional array of small picture elements called *pixels*. Patterns are constructed by turning some of these pixels on, some off.

For the Macintosh, this pixel array is 512 across by 342 down. Each pixel corresponds to a unique single bit in the computer's memory. To change the appearance of the pixels on the screen, you change the binary values stored in these bits of memory. A binary value of zero for a particular bit makes the corresponding pixel turn white, a binary value of one makes the pixel turn black. This is opposite of many computer screens, but it makes the Macintosh's screen look more like a normal sheet of paper with dark "ink" on a white background.

A *pattern* is an eight-by-eight array of pixels that is repeatedly laid down on the screen in this larger pixel array whenever you draw, fill, or erase. Although patterns are geometric, they are stored in memory as numbers whose bit values correspond to the white/black color of the pixels.

QuickDraw uses the following data structure for a *pattern*:

```
Pattern = PACKED ARRAY [0..7] OF 0..255;
```

This means that a pattern requires eight bytes of memory.

Patterns can be defined by individually loading the eight bytes of this array, or they can be defined using QuickDraw's "StuffHex" routine, which allows you to see each bit of pattern as you load it. It is also possible to store patterns as resources.

In this section, we learn how to "stuff" patterns. The routine "StuffHex" expects two parameters: a pointer and a string of hexadecimal digits. The routine converts this string into a sequence of binary numbers that it places in memory starting at the address indicated by the pointer.

Our goal is to design and load a pattern using this procedure. We start by marking off an eight-by-eight block on graph paper and filling in

the squares in this block based on what we want to see in our pattern. Then we convert each row to a corresponding pattern of ones and zeros. Each row forms an eight-digit binary number, which we write as a two-digit hexadecimal number. We "pack" these eight hexadecimal numbers (in top to bottom row order) to form a 16-digit number. The digits thus form a string that we can use in the StuffHex routine to define the pattern for the Macintosh (see Figure 4-5). This pattern serves as the background for our next example program.

Defining Cursors

The cursor shape reminds the user of the current "mode"; that is, which options are currently available to the user. As mentioned in Chapter 1, the developers of the Macintosh made every effort to avoid modes. However, in spite of those efforts, modes are inevitable. For example, the MacPaint program has a number of drawing modes, including pattern editing, "lasso" mode, line drawing, and erasing. These modes are indicated by a variety of cursor shapes, including an arrow, a pencil, a paint bucket, a cross-hair, and a lasso (see Figure 4-6).

Figure 4-5. Designing a Pattern

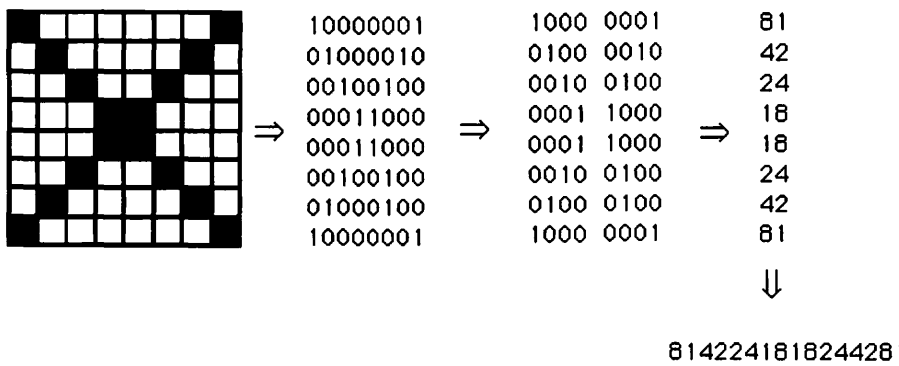
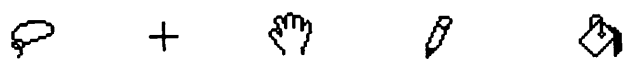


Figure 4-6. Cursor Shapes



In this section, you see how the cursor is stored and how to define and switch among your own cursor shapes.

Let's see how a cursor is stored. The cursor on the Macintosh's screen resides in a data structure called "cursor", defined in the "QuickDraw" UNIT as:

```
Cursor = RECORD
           data:    Bits16;
           mask:    Bits16;
           hotSpot: Point;
         END;
```

where "Bits16" is defined as:

```
Bits16 = ARRAY[0..15] OF INTEGER;
```

Both the ".data" and ".mask" fields are of type "Bits16", which is a 16 by 16 array of bits in memory that maps to a 16 by 16 array of pixels on the screen. Notice that these cursor arrays are twice as big horizontally and vertically as the pattern array just studied. This larger size allows the cursor shape to have enough resolution to make interesting pictures, such as those used by MacPaint. Other pattern arrays are smaller, so they are repeated often in order to be recognizable even when they are "painted" into small areas on the screen.

The cursor has two image arrays to indicate how it should cover what is on the screen. For each bit position, the mask and data interact according to the following table:

If the mask bit is zero, the data bit determines whether or not the pixel is unchanged or reversed. If the mask bit is one, then the data bit determines whether the pixel is white or black.

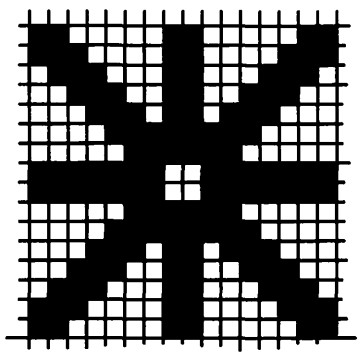
Table 4-1. Data and Mask for Cursors

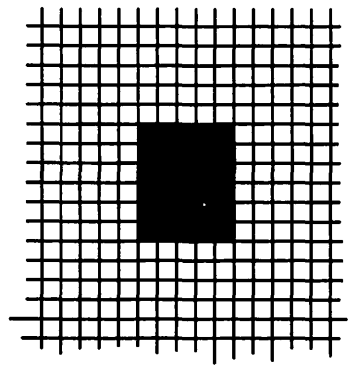
Mask	*Data*	*Result*
0	0	Same as original
0	1	Inverse of original
1	0	White
1	1	Black

The data and masks for cursors can be designed on graph paper in the same way that patterns are designed. Figure 4-7 shows the mask and data bit arrays for a "spider" cursor that we use in our last example program.

The third field, ".hotSpot", specifies the hot spot for the cursor. This is the point, such as the *tip* of the arrow cursor or pencil cursor, where the action is. For example, with the pencil cursor, the "hot spot" tip indicates the position of the pixel to be changed. With the arrow cursor, you place the hot spot within a region to select the corresponding object.

Figure 4-7. The Mask and Data for the Spider Cursor





Data

Binary	Hex
1100 0001 1000 0011	C183
1110 0001 1000 0111	E187
0111 0001 1000 1110	718E
0011 1001 1001 1100	399C
0001 1101 1011 1000	1DB8
0000 1111 1111 0000	0FF0
0000 0111 1110 0000	07E0
1111 1110 0111 1111	FE7F
1111 1110 0111 1111	FE7F
0000 0111 1110 0000	07E0
0000 1111 1111 0000	0FF0
0001 1101 1011 1000	1DB8
0011 1001 1001 1100	399C
0111 0001 1000 0111	718E
1110 0001 1000 0111	E187
1100 0001 1000 0011	C183

Mask

Binary	Hex
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0111 1110 0000	07E0
0000 0111 1110 0000	07E0
0000 0111 1110 0000	07E0
0000 0111 1110 0000	07E0
0000 0111 1110 0000	07E0
0000 0111 1110 0000	07E0
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000
0000 0000 0000 0000	0000

The Pattern Example

Our next example program, called "Pattern", illustrates patterns and cursors.

The program opens with a crosshatched screen that we developed (see Figure 4-8). The cursor has the "spider" shape as described previously. As you move this cursor, it interacts with the crosshatched pattern in the background to create a crawling effect.

Here is the program:

```
PROGRAM Pattern;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf;
```

Figure 4-8. The Pattern Program

```
PROCEDURE ClickButton;
   BEGIN
      WHILE Button DO;
      WHILE NOT Button DO;
      WHILE Button DO;
   END;

PROCEDURE SetUpSys;
   BEGIN
     InitGraf(@thePort); {Initialize QuickDraw default variables}
     InitFonts;          {Initialize Font manager}
     NEW(thePort);       {Set up grafPort}
     OpenPort(thePort);
   END;

PROCEDURE MakePat;
   VAR
     Pat : Pattern;
     Cur : Cursor;

   BEGIN
     StuffHex(@Pat, '8142241818244281');
     FillRect(thePort^.portBits.Bounds, Pat);

     StuffHex(@Cur.data,
'C183E187718E399C1DB80FF007E0FE7FFE7F07E00FF01DB8399C718EE187C183');
     StuffHex(@Cur.mask,
'00000000000000000000007E007E007E007E007E007E0000000000000000000000');
     SetPt(Cur.hotSpot, 8, 8);
     SetCursor(Cur);
   END;

BEGIN {main program}
   SetUpSys;
   MakePat;
   ClickButton;
END.
```

Data Structures

The "Pattern" program has the standard USES section.

Procedures

The "Pattern" program has a number of procedures, including the "ClickButton" procedure that appeared in the first program.

"SetUpSys" Procedure. The procedure "SetUpSys" contains some of the same initialization steps that comprised the main part of the last program. In future programs, we place all initialization steps in a routine of this name.

Making Patterns and Cursors. The "MakePat" procedure sets up the crosshatched pattern on the screen and the spider cursor. It has two local variables: a pattern, "Pat", and a cursor, "Cur".

The procedure begins by calling the built-in "StuffHex" routine to specify the pattern. We place "@Pat" in its first parameter to point to the pattern "Pat", declared as one of its local variables. We place the desired string of hexadecimal digits in the second parameter.

The procedure then calls "StuffHex" two more times: once to define the ".data" field of our cursor, and once to define its ".mask" field. We also call "SetPoint" to define the location of the hot spot. We then call "SetCursor" to install the cursor.

The Main Program

The main part of the program calls our three routines in sequence. First, it calls "SetUpSys" to perform the proper initializations. Second, it calls "MakePat" to draw the new background pattern and set the spider cursor. Third, it calls "ClickButton" to pause, waiting for the user to hit the mouse button before the program exits.

Notice that the program contains no commands to update the cursor as the mouse is moved by the user. Instead, this is automatically done by the Macintosh's Operating System as a background task. That is, every time the mouse is moved, interrupts are generated that update the coordinates of its position, and periodically (as often as 60 times a second) the system automatically redraws the mouse on the screen.

Memory-Mapped Video

Now let's study QuickDraw's video screen and data structures. We begin with *memory-mapped video*, *bitImages*, and *bitMaps*. Later, we introduce *coordinate systems*, *points*, *rectangles*, and *regions*, concepts that form shapes of increasing complexity.

Pixel Array

As we have shown, the Macintosh's video screen can be viewed as a two-dimensional array of *pixels*. The position of each pixel is described by two indices that correspond to its row and column position. Both indices

run from zero, and the index value of [0,0] specifies the pixel in the upper left corner of the screen. A value [i,j] indicates the pixel in the ith column and jth row of the screen. That is, the first index gives the horizontal position, and the second gives the vertical position, of the pixel.

Video RAM

The bits that map to the screen reside in two areas of memory in the Macintosh called video RAM. The exact location of these areas depends on the amount of memory in your Macintosh. On the 128K Macintosh, the primary area starts at $1A700 (hexadecimal) and a secondary area starts at $12700. On the 512K Macintosh, these are at $7A700 and $72700, respectively. In either case, both areas contain 21,888 bytes. Normally, the primary area is used, but there is a bit in the Mac's memory that switches back and forth between these two areas. This requires some special programming because the secondary area is normally for other purposes.

As with most microcomputer memory, the bits in the Macintosh's video memory are organized into bytes. Thus, video memory behaves just like regular memory. However, this adds an extra level of complexity to the mapping between the pixel array and the video RAM. That is, each bit is identified by a byte *address* and a *bit position* within that byte, whereas pixels are identified by their horizontal and vertical position numbers.

Raster Scanning

To understand this mapping, it helps to know how the video hardware inside the Macintosh works. The video hardware constantly scans the video RAM, converting its digital information into a video signal that produces the picture on the screen. As the hardware scans the memory, an electron beam scans the screen in what is called a "raster" pattern — a series of closely spaced horizontal lines.

Simultaneously scanning the memory and the screen creates the mapping from the memory to the screen. Each bit in memory maps to the unique pixel position on the screen where the beam is located when that bit is scanned.

In the Macintosh, bytes are scanned in the increasing order of their addresses, from the beginning of video memory to its end. Within each byte, the bits are scanned starting with bit 7 and decreasing in order to bit 0 (see Figure 4-9).

As a result, each byte of video memory maps to a horizontal row of eight pixels on the screen, with bit 7 on the left and bit 0 on the right (see Figure 4-10).

The first byte of video RAM maps to the eight pixels on the left of the top row of the screen. The next byte maps to the next eight pixels to the right, and so on. A total of 64 bytes maps in this way to the top row of the screen. Each successive 64 bytes of video memory map to a new row of the screen in consecutive order. Thus, the 21,888 bytes of video RAM are divided into 342 continuous sections of 64 bytes, one section for each row of display.

Formulas

The corresponding bit for the pixel position [i,j] sits in a byte whose address in video RAM depends on the row and column indices according to the following formula:

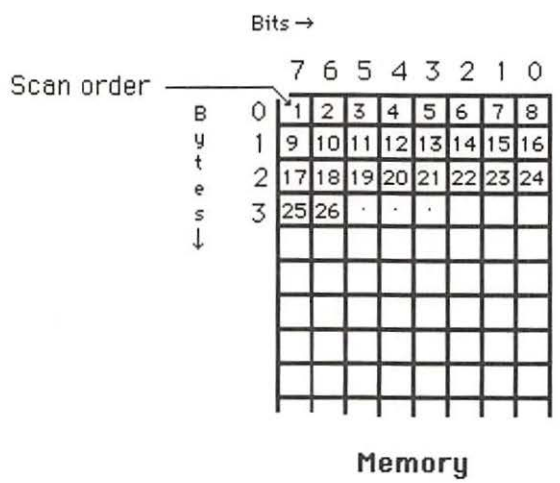
```
Base Address + 64*j + int(i/8)
```

The row index j is multiplied by 64 because there are 64 bytes of video RAM per row. The column index is divided by eight because there are eight pixels per byte. Note that 64 is the row length for the Macintosh screen, so this number figures significantly in the mapping formula.

The formula:

```
7 - i mod 8
```

Figure 4-9. Scanning the Bytes of Memory



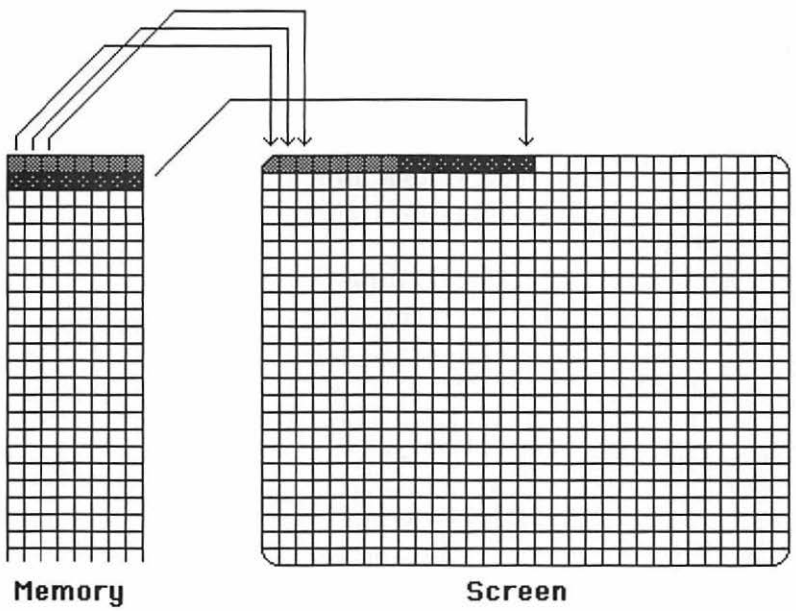
gives the bit position within the byte. The term "i mod 8" gives the remainder when the horizontal index i is divided by the number of pixels per byte (that is, eight).

BitImages

A *BitImage* is a simple, yet elegant, Quickdraw data structure corresponding to the organization of video RAM. Its information indicates how an area of memory holds images. This leads to the concept of *BitMap*, which controls QuickDraw operations at the most primitive level.

Technically, a *BitImage* is an area of memory defined by a base address (first address) and a row length measured in bytes. In general, the base address can be anywhere in memory and the row length can be any non-negative integer. However, for the Macintosh screen, the base address is the beginning of video memory and the row length is equal to 64 bytes. BitImages with other base addresses can store pictures for later display on the screen, and BitImages with other row lengths can build images for display on other devices, such as printers or flat-screen displays.

Figure 4-10. Mapping Bits to the Screen



BitMaps

QuickDraw never declares "BitImage" as a data type. Instead, it incorporates the basic information of a "BitImage" in the data structure called "BitMap". The default variable "screenBits" is of this type, as is the ".portBits" field of a grafPort. Here is the "official" Pascal definition of a "BitMap":

```
BitMap = RECORD
     BaseAddr: Ptr;
     RowBytes: INTEGER;
     Bounds:   Rect;
END;
```

The first field, ".BaseAddr", is of type "Ptr". This stores the address of the first byte of the screen and locates the corresponding BitImage in memory. The second field, called "RowBytes", is of type INTEGER and defines the mapping. It gives the number of bytes per row. Multiplying this by eight gives the number of pixels per row on the real or imaginary screen. The last field, called "Bounds", is of type "Rect". It gives the corners of this screen, thereby delimiting the picture.

It is important to realize that the BitMap does not contain the BitImage (actual data on the screen). It merely points to and describes the data. That is, it contains the starting address, the size, and the number of bytes per row of this area, but not the contents of any bytes in it.

Coordinate Systems

In the last section, we saw the screen as a doubly indexed array of pixels. We now place a coordinate system on it to define "points", "rectangles", and "regions". The coordinate system allows us to address points more easily and provides minimum and maximum horizontal and vertical limits for our screen images.

At the lowest level, all such coordinates are stored as type INTEGER. Recall that the type INTEGER is stored in the Macintosh as ordinary signed 16-bit integers. As a result, any Quickdraw coordinate value ranges from –32,768 to 32,767 and can take on only integer values.

QuickDraw approaches geometric objects differently than it does pixels on the screen. The difference is subtle but important. Points are considered zero-dimensional objects; that is, they are infinitesimally thin, with zero width and height. On the other hand, pixels are two-dimensional; that is, they have unit width and height (see Figure 4-11).

QuickDraw has a coordinate system for the screen that closely corresponds to the pixel indexing. The horizontal axis runs across the top of the screen from left to right. The vertical axis runs along the left side of the screen from top to bottom. The units for the coordinates in each direction are measured in pixels (see Figure 4-12).

As a result, the point with coordinates (i,j) always falls on the upper corner of the pixel indexed by [i,j]. In particular, the upper left corner of the screen has coordinates (0,0) (see Figure 4-12). Please note the use of round brackets for coordinates and square brackets for indices.

Each QuickDraw graphics entity is defined by its own data structure, which describes how its coordinates are stored in the Macintosh's memory and accessed by the programmer. This chapter examines how this applies to points, rectangles, and regions.

Figure 4-11. Points and Pixels

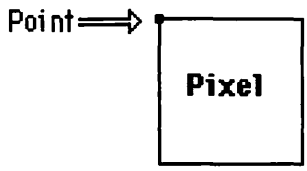
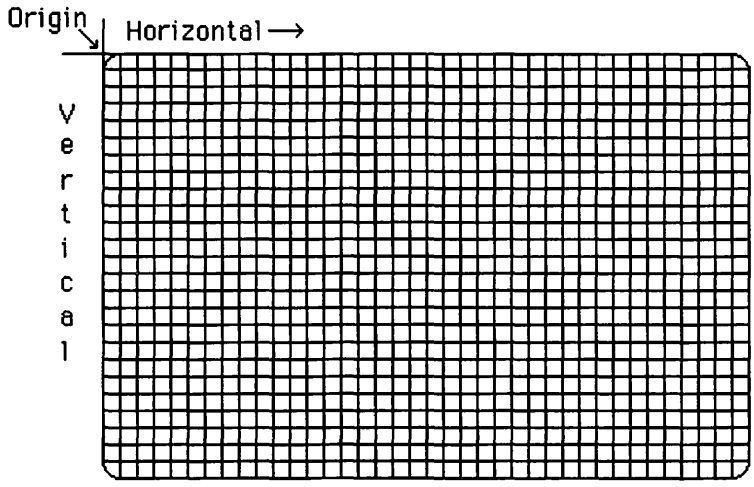


Figure 4-12. The Coordinate System



Points

Let's look at the simplest graphics entity, the point. The point forms the basis of other shapes, such as rectangles and regions.

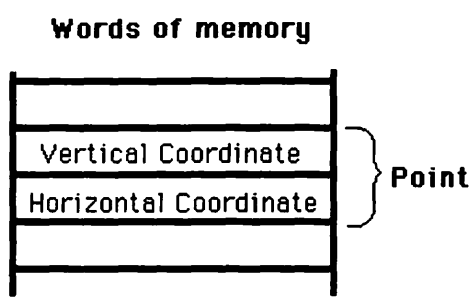
Mathematically, a point is described by a coordinate pair (x,y). For QuickDraw, this means that a point requires two 16-bit words of memory (see Figure 4-13). The first contains the vertical coordinate (y coordinate), the second contains the horizontal coordinate (x coordinate). This is how the assembly language programmer views points.

The Pascal programmer must deal with two ways to access points, creating an additional layer of complexity. Both ways are directly related to and are, in fact, equivalent to the low-level description. Two methods give the programmer more freedom. However, we normally use neither method, referring instead to points without getting into their internal structure.

In the first method, a point is regarded as a *record* consisting of two *fields*, one for each coordinate. If Pt is a QuickDraw Pascal variable of type "Point", then Pt.v is the vertical coordinate (y coordinate), and Pt.h is the horizontal coordinate (x coordinate).

The second method is more complicated. Here, the point is regarded as a *record* with one *field*, which is an array of two integers. That is, if "Pt" is a QuickDraw Pascal variable of type "Point", then Pt.vh[v] is its vertical coordinate and Pt.vh[h] is its horizontal coordinate. Here, "v" and "h" are identifiers belonging to a QuickDraw scalar type called "VHSelect". This method allows the programmer to use FOR loops and the like to index through the coordinates of the point. However, only two coordinates hardly make it worth the effort.

Figure 4-13. Internal Storage of Points



These structures are built into the software as either an integral part of the software (in the case of the "Instant" Pascal interpreter), or external files that must be "used" (in the case of the Pascal compiler).

For the Pascal compiler, the QuickDraw UNIT contains the following TYPE declarations:

```
VHSelect = (v,h);

Point = RECORD CASE INTEGER OF
     0: (v: INTEGER; h: INTEGER);
     1: (vh: ARRAY [VHSelect] OF INTEGER)
END;
```

Experienced Pascal programmers will recognize that "Point" is a variant record structure; that is, there are two ways to access this structure. These correspond to the preceding descriptions. The type "VHSelect" is defined first. It is a scalar type consisting of two identifiers, "v" (for vertical) and "h" (for horizontal). This assists in defining the second variant form. Next, "Point" itself is defined either as two integer fields, ".v" and ".h", or as one field, ".vh", which is an array of two integers.

Mouse Point Example Program

Now that we understand "Point", let's look at a Pascal program that plots mouse points on the screen using the formulas developed earlier.

In this example we use both forms of the data structure for a point, though this is not good programming practice unless special reasons call for mixing them.

When this program is run, we first see a completely white screen. If we move the mouse, we see a trail of darkened pixels where the mouse was (see Figure 4-14). If the mouse is moved rapidly, the pixels are far apart; if the mouse moves slowly, the pixels tend to form a continuous trail. We have "hidden" the mouse cursor to prevent the cursor from interfering with the pixels being plotted. You can exit the program by moving the mouse point to the top of the screen.

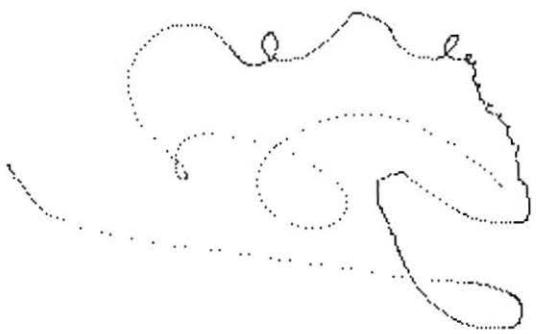
```
PROGRAM Pt;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    }Memtypes,
      {$U obj/QuickDraw   }QuickDraw,
      {$U obj/OSIntf      }OSIntf,
      {$U obj/ToolIntf    }ToolIntf;
```

```
VAR
   MousePt: Point;

PROCEDURE SetUpSys;
   BEGIN
      InitGraf(@thePort);
      NEW(thePort);
      OpenPort(thePort);
      EraseRect(thePort^.portBits.bounds);
      HideCursor;
   END;

PROCEDURE PutMouse(thePt: Point);
   BEGIN
      BitSet(POINTER(ORD(screenBits.baseAddr)
                     + screenBits.rowBytes * thePt.v
                       + thePt.h DIV 8),
             thePt.vh[h] MOD 8);
   END;

BEGIN  {main program}
   SetUpSys;
   REPEAT
      GetMouse(MousePt);
      PutMouse(MousePt);
   UNTIL MousePt.vh[v] < 20;
END.
```

Data Structures

This program has the same USES section as our first program, allowing us access to the various data types, data variables, and procedures used by QuickDraw and other parts of the Macintosh's built-in software.

Figure 4-14. Output of the Mouse Point Program



The VAR section of this program contains one variable — "MousePt", of type "Point". The type "Point" is defined in the UNIT "QuickDraw".

"SetUpSys" Procedure

The procedure "SetUpSys" initializes the system for this program. It contains some of the same initialization steps as the "SetUpSys" routine of the last program. However, it also contains a step to erase the screen, a task that was not necessary in the last program because the screen was filled with a pattern immediately after the "SetUpSys" routine was called. In the present program, we erase the screen using the "EraseRect" procedure on the ".bounds" field of the ".portBits" field of the grafPort. The ".portBits" field is of type "BitMap", discussed earlier.

We also call "HideCursor" so that the cursor does not interfere with the workings of our program. In future programs, we place all initialization steps in a routine of the same name, "SetUpSys", that we used for this routine.

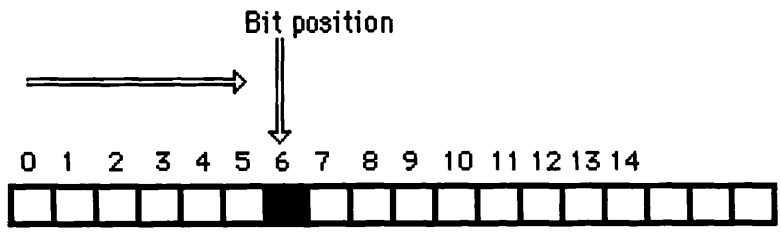
"PutMouse" Procedure

The procedure "PutMouse" plots a pixel whose position is given. It expects one parameter of type "Point", which specifies the location of the pixel.

The procedure uses the "BitSet" procedure. "BitSet" expects two arguments. The first argument must be of type "Ptr" and should point to a location in memory. The second argument of BitSet is of type Long Integer and gives the bit position, counting from left to right with bit position 0 on the left.

The "BitSet" routine makes the bit in the specified bit position of memory equal to one, leaving the other bits in memory alone (see Figure 4-15).

Figure 4-15. BitSet Positions



The "BitSet" routine in ROM is very clever. It is not affected by byte (8-bit) versus word (16-bit) or even long word (32-bit) addressing considerations. It uses the BSET processor instruction, which acts only on bytes when it addresses memory; yet the ROM routine properly handles very large bit positions. To accomplish this, the ROM routine uses some tricky code that involves a special indexing addressing mode.

Let's look at the two parameters of "BitSet" in more detail. First, look at the type "Ptr" in the first parameter. The type "Ptr" is defined in the TYPE section of "Memtypes" by the following two type statements:

```
SignedByte = -128..127;
Ptr = ^SignedByte;
```

To the Pascal compiler, this means that type "Ptr" parameters can point to any byte in memory. Recognizing this, the Pascal compiler uses the byte addressing mode when translating memory fetched by this pointer.

Note that Macintosh Pascal does not interpret this declaration in the same way. It treats a variable declared in this way as a pointer to integers and uses the 16-bit word addressing mode when it accesses data using such a pointer. This can be disastrous when the address contained in the pointer is an odd number: the MC68000 processor does not accept odd addresses for 16-bit words. In fact, when the processor senses an odd address while in word mode, it interrupts and goes to a special error routine maintained by the Operating System.

In the expression for the first parameter, we use the formula developed earlier to compute the address of the appropriate memory location. We use the fields ".baseAddr" and ".rowBytes" of the default variable "screenBits" and the ".v" and ".h" fields of our own "Point" parameter. We use the first variant form of the data type "Point" to individually grab the horizontal and vertical coordinates of the point.

In the formula, the ORD function converts the pointer "screenBits.baseAddr" to a long integer, and the POINTER function converts the long integer address back to a pointer.

Now let's look at the second parameter. Notice that the BitSet routine numbers the bits opposite to their usual order. This simplifies the formula for plotting pixel bits. We use the formula:

```
thePt.vh[h] MOD 8
```

instead of the reversed formula:

```
7 - thePt.vh[h] MOD 8
```

that we presented above.

At this stage of the program, the second variant form for "Point" (".vh[h]") is used with "thePt" in our BitSet command. This simply illustrates that particular variant form. Normally, you should use the first, more straightforward variant form (".h").

The Main Program

The main program begins by calling the "SetUpSys" procedure to initialize QuickDraw.

Next is the REPEAT loop where the program goes round and round, getting the mouse position and plotting it on the screen. The first statement calls the ROM routine "GetMouse" to determine the position of the mouse; then a second statement routine calls our "PutMouse" to plot the corresponding pixel.

The UNTIL statement at the bottom of the REPEAT loop monitors the vertical position of the mouse, continuing the loop as long as the mouse stays below the menu bar that occupies the top 20 rows of the screen. Again, the second variant form for "Point" is for illustration.

Rectangles

A level of complexity above points are *rectangles*. They describe shapes such as ovals and rounded rectangles and are used in the structure of a region, which is fundamental to Macintosh's overlapping windows.

This section describes a QuickDraw rectangle as a variant record structure of four coordinate variables or two "Points".

In computer graphics, a rectangle is a four-sided figure whose sides are always parallel to the coordinate axes. A rectangle is defined by its four corner points (x1,y1), (x1,y2), (x2,y1), and (x2,y2) (see Figure 4-16). It can be thought of as the set of points whose x coordinate lies between x1 and x2 and whose y coordinate lies between y1 and y2.

In a QuickDraw rectangle, the point (x1,y1) is placed at the upper left corner of the rectangle, point (x1,y2) is the lower left corner, (x2,y1) is the upper right corner, and (x2,y2) is the lower right corner of the rectangle. In consideration of this arrangement, QuickDraw refers to x1 as "left", x2 as "right", y1 as "top", and y2 as "bottom".

For example, the screen itself is in the rectangle given by:

left = 0

right = 512

top = 0

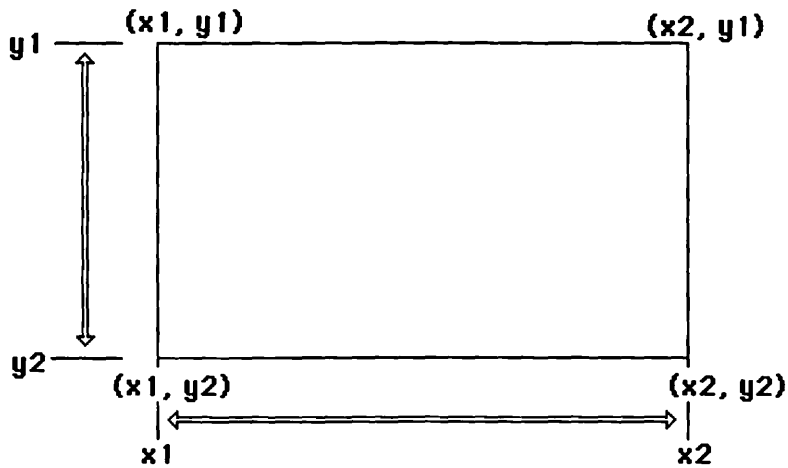
bottom = 342

Notice that (x,y) are the coordinates of points in the rectangle, not pixel indices. Pixel indices would be bounded by 511 and 341, not 512 and 342 (see Figure 4-17). Also notice that the coordinates have a definite order: in this example, the value of "left" is less than the value of "right", and the value of "top" is less than the value of "bottom". In general, if "left" is greater than or equal to "right", or if "top" is greater than or equal to "bottom", then no points can satisfy the inequalities. In this case, we say the rectangle is *empty*.

QuickDraw also specifies rectangles by the pair of opposite corners at the top left and bottom right of the rectangle (see Figure 4-18). The top left corner is called topLeft and has coordinates (left,top). The bottom right corner is called botRight and has coordinates (right,bottom). This specifies the same information as before, only in a slightly different format. Again, having two ways of doing something gives more freedom, allowing the programmer to use the more convenient form.

Look at the example of the entire screen again. It is defined by the opposite corner points (0,0) and (512,342) (see Figure 4-19). Notice that

Figure 4-16. Defining Rectangles



the point topLeft, with coordinates (0,0), is the upper left corner of the upper left pixel of the screen; the point botRight, with coordinates (512,342), is the lower left corner of its lower left pixel.

The Pascal data structure for the type "Rectangle" is as follows:

```
Rect = RECORD CASE INTEGER OF
     0 : (top:    INTEGER;
          left:   INTEGER;
          bottom: INTEGER;
          right:  INTEGER);
```

Figure 4-17. Bounding Rectangles

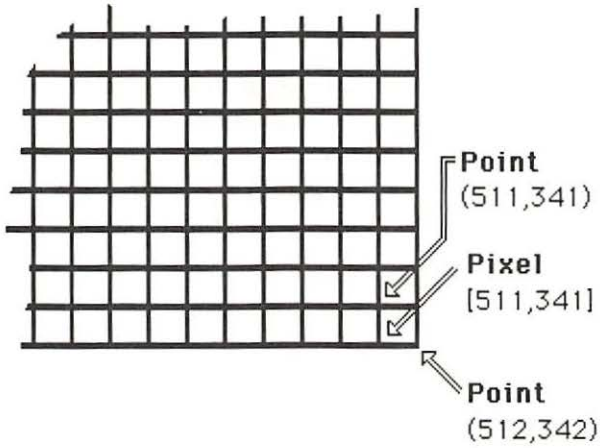
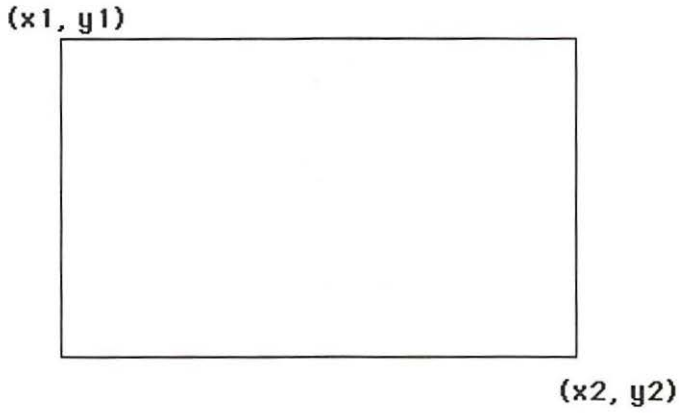


Figure 4-18. Opposite Corners Define a Rectangle



```
1 : (topLeft:  Point;
     botRight: Point)
END;
```

Again, QuickDraw uses a variant record structure. In the first form, the four-integer coordinate values that define the rectangle are listed in the following order: top, left, bottom, right. In the second form, the two opposite *points* that define the rectangle are listed.

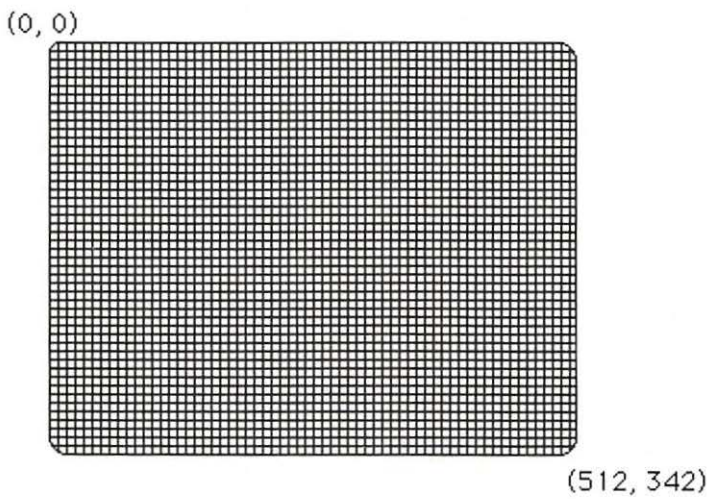
It is important to note that the forms are equivalent and lead to the same internal storage of relevant coordinate information. To an assembly language programmer this structure is just a sequence or list of four 16-bit integers (see Figure 4-20).

An Example of Rectangles

The following example of a Pascal program illustrates rectangles in their various forms (see Figure 4-21). The program defines three rectangles — a large square; a low, wide rectangle; a high, thin rectangle — then uses them to draw a checkerboard pattern on the screen. After each rectangle is drawn, you must click the mouse button to continue.

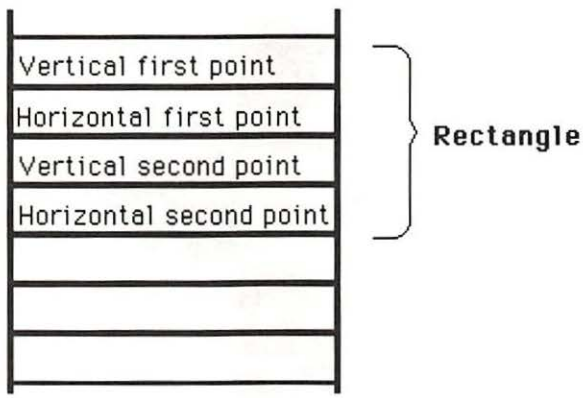
```
PROGRAM Rectangle;
   {$R-}{$X-}
```

Figure 4-19. The Screen Limits



```
USES
    {$U obj/Memtypes    }Memtypes,
    {$U obj/QuickDraw   }QuickDraw,
    {$U obj/OSIntf      }OSIntf,
    {$U obj/ToolIntf    }ToolIntf;

VAR
    Square, Wide, Tall: Rect;

PROCEDURE ClickButton;
    BEGIN
        While Button DO;
        While NOT Button DO;
        While Button DO;
    END;

PROCEDURE SetUpSys;
    BEGIN
        InitGraf(@thePort);
        NEW(thePort);
        OpenPort(thePort);
        EraseRect(thePort^.portRect);
        InitCursor;
    END;

BEGIN  {main program}
    SetUpSys;

    Square.left := 100;
    Square.top := 30;
```

Figure 4-20. Internal Storage of Rectangles



```
    Square.right := 300;
    Square.bottom := 230;
    FrameRect(Square);
    ClickButton;

    Wide.topleft.h := 100;
    Wide.topleft.v := 100;
    Wide.botright.h := 300;
    Wide.botright.v := 160;
    PaintRect(Wide);
    ClickButton;

    Tall.topleft.vh[h] := 170;
    Tall.topleft.vh[v] := 30;
    Tall.botright.vh[h] := 230;
    Tall.botright.vh[v] := 230;
    InvertRect(Tall);
    ClickButton;
END.
```

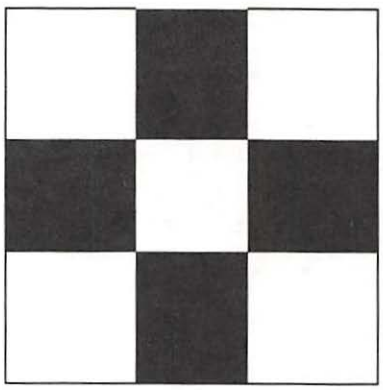
Data Structures

The USES section is the same as in the last compiler Pascal program. The VAR section declares the three rectangles "Square", "Wide", and "Tall" to be of type "Rect".

Procedures

This program uses the same "ClickButton" and "SetUpSys" procedures that appeared in previous programs.

Figure 4-21. The Rectangles



The Main Program

In the main part of the program we specify the coordinates of these three rectangles. We demonstrate different ways to describe each rectangle. The square is specified by the first variant form, which directly addresses each coordinate: left, top, right, and bottom. The wide and the tall rectangles are specified by the second variant form of "Rect", in which two opposite corner points are specified. The wide rectangle uses the first variant form of "Point", and the tall rectangle uses its second variant form. This gives all possible variant forms of "Rect".

The first rectangle is "framed" using the "FrameRect" QuickDraw routine. The second rectangle is "painted" (filled with black) using the "PaintRect" QuickDraw routine, and the third rectangle is "inverted" (white changes to black and vice versa) using the "InvertRect" QuickDraw ROM routine. After each rectangle is drawn, we call "ClickButton" to pause the display.

In practice, the QuickDraw routine "SetRect" is more efficient to specify the corners of a rectangle. It expects five parameters: a rectangle and the four coordinates that delimit it. We see examples later.

Regions

Regions are important as the fundamental building blocks used by the *Window Manager* to draw and manage window parts such as the *goAway box*, the *grow area*, and the *drag region*; and for the *Control Manager* to draw control parts such as *scroll bars*, *check boxes*, and *buttons*. *Regions* are also used by the *Window Manager* to control visibility and updating when windows overlap.

Regions have been carefully designed so that these window and control parts can be quickly drawn on the screen, then found with the mouse in any shape they take on. Understanding how regions work is the first step in understanding how the Macintosh's interactive programming facilities work.

In this section we will study *regions*, an essential part of the picture-making environment in which QuickDraw operates. Regions control the visibility of objects drawn on the screen and can define and display complicated shapes.

This section discusses the internal structure of regions. An example program demonstrates how to display the internal structure of any region.

Regions and Data Structures

A region is a dynamic data structure. That is, at different times a particular region may contain different amounts of data. To help manage this, each region contains an integer called "rgnSize" that tells how many bytes of data it currently contains.

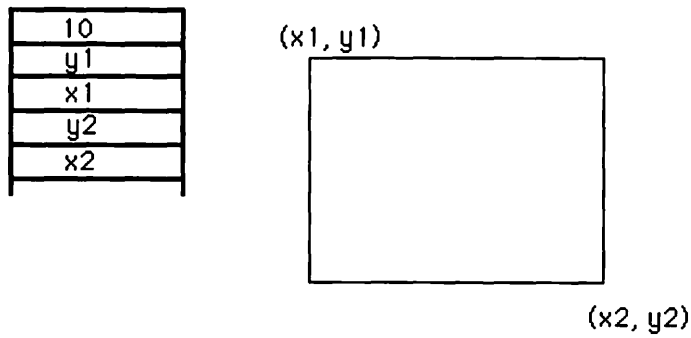
The simplest region contains 10 bytes and has the shape of a rectangle (see Figure 4-22). We call these regions rectangular. In this case, the 10 bytes form five 16-bit integers, the first of which is "rgnSize"; the remaining four are corner coordinates of this rectangle, called the "rgnBBox".

In general, a region can have any shape that can be drawn within the QuickDraw coordinate system (with its 16-bit integer coordinates) (see Figure 4-23). If this shape is not a rectangle, then we call the region nonrectangular.

Nonrectangular regions contain the 10 bytes described above, plus additional bytes that give coordinate information about the "corners" of a shape. In this case, the *rgnSize* is greater than 10 and the *rgnBBox* is a rectangle that surrounds the shape, a description that aptly fits its full name, *region boundary box*.

The data that defines the shape of a nonrectangular region is organized in an ingenious way. It is a list of integers that makes up a sequence of fields, each of which specifies the corner points of the shape that lie along one horizontal line in QuickDraw's coordinate system. Each field begins with a y coordinate that uniquely specifies the height of the horizontal line, followed by a sequence of x coordinates that specify the horizontal positions of its corner points. Each field terminates with an integer whose value is 32767, and the entire list terminates with an extra

Figure 4-22. Rectangular Regions



integer with the value of 32767. The fields are given in increasing order of their y coordinates, and the x coordinates within each field are also given in increasing order.

Here is an example of the data for a simple L-shaped region (see Figure 4-24). First, it is shown as a list of integers. This is how it would be stored in the Macintosh's memory:

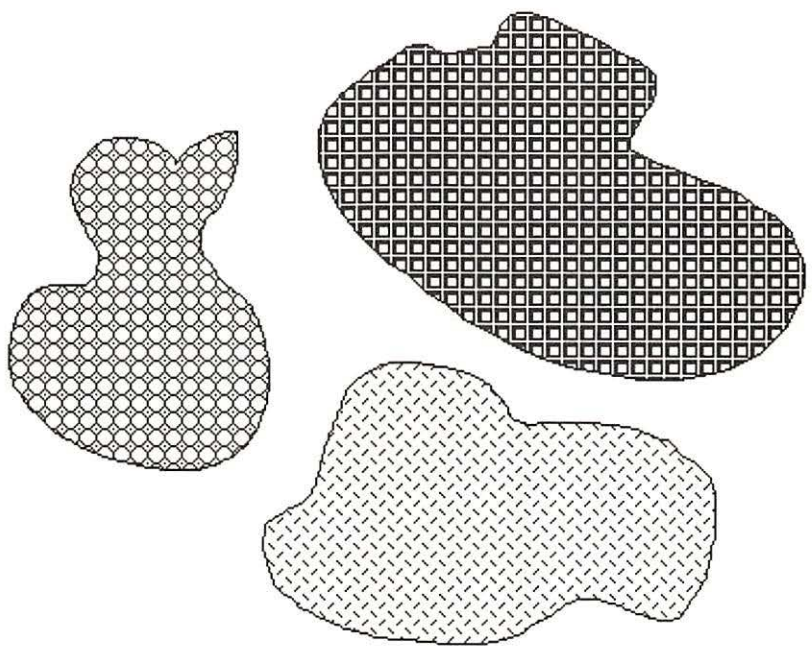
```
100, 100, 200, 32767, 200, 200, 300, 32767, 300, 100, 300, 32767, 32767
```

Now let's break it into fields, using the value 32767 as a terminator for each field.

```
100, 100, 200, 32767,
200, 200, 300, 32767,
300, 100, 300, 32767,
32767
```

Finally, let's extract the coordinates of each corner point of this shape. Remember that the first integer of each field is the y coordinate and the

Figure 4-23. Nonrectangular Regions



subsequent integers are x coordinates. Look carefully to see where each coordinate fits.

```
(100,100),  (200,100)
(200,200),  (300,200)
(100,300),  (300,300)
```

Keeping Track of a Region's Data

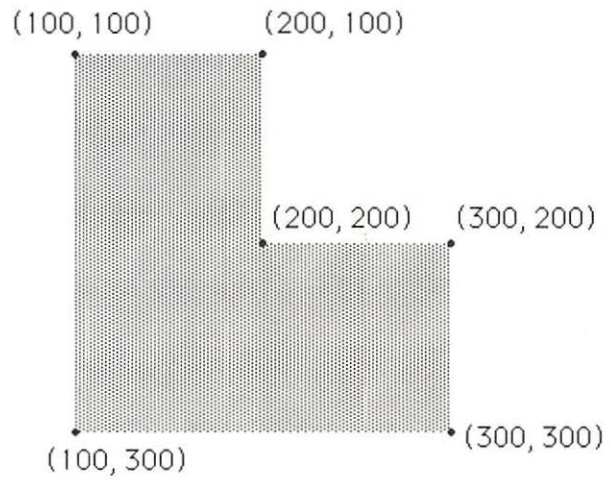
A region is a dynamic structure: it moves in memory as it grows or shrinks or as computations are performed on it. To keep track of where the data are currently located, regions are accessed through a series of pointers.

Each region is normally accessed through a pointer called the *region handle*. The region handle does not point directly at the region but rather to another pointer called the *region pointer*, which points to the actual data. In Chapter 3, we discussed the reasons for such a system.

To the Pascal programmer, a region is defined by the following set of Pascal TYPE statements in the external QuickDraw file:

```
RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;
Region    = RECORD;
              rgnSize: INTEGER;
              rgnBBox: Rect;
            END;
```

Figure 4-24. An L-Shaped Region



Here we see that type *RgnHandle* is a pointer to objects of type *RgnPtr*, that type *RgnPtr* a pointer to objects of type *Region*, and that type *Region* a record structure containing fields for "rgnSize" and "rgnBBox" (see Figure 4-25).

Note that the "nonrectangular" part of the data is not declared because it is managed by the QuickDraw routines. Pascal needs to know only the total size of the data area.

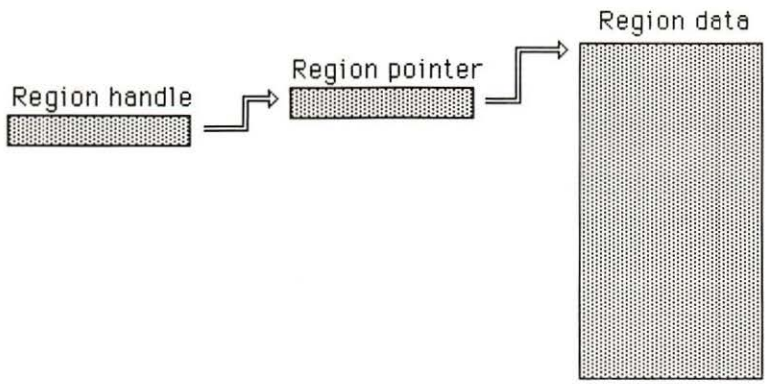
The Region Program

Here is a Pascal program that defines the preceding region in a *natural* way by "unioning" two rectangular regions, then displaying the corner points of this figure (see Figure 4-26).

This kind of exploratory program might be written to discover how something complex works in the computer. It contains a section to generate an object in a normal way and a procedure to pick this structure apart. In this case, use the "ShowCorners" procedure to peek at any region of your own design, produced by any method.

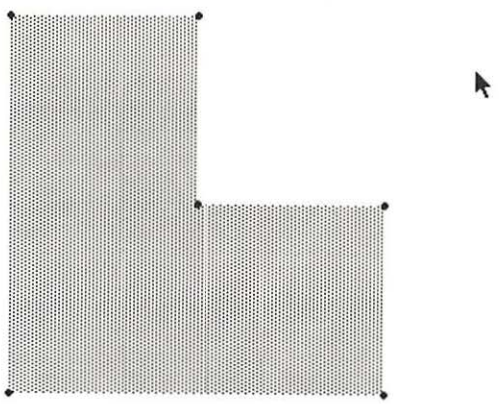
```
PROGRAM Region;
   { $R-}{$X-}

   USES
      {$U obj/Memtypes    }Memtypes,
      {$U obj/QuickDraw   }QuickDraw,
      {$U obj/OSIntf      }OSIntf,
      {$U obj/ToolIntf    }ToolIntf;
```

Figure 4-25. Region Pointers and Handles



```
VAR
   RgnA, RgnB : RgnHandle;

PROCEDURE ClickButton;
   BEGIN
      While Button DO;
      While NOT Button DO;
      While Button DO;
   END;

PROCEDURE SetUpSys;
   BEGIN
      InitGraf(@thePort);
      NEW(thePort);
      OpenPort(thePort);
      EraseRect(thePort^.portRect);
      InitCursor;
   END;

PROCEDURE ShowCorners (theRgn : RgnHandle);
   TYPE
      numArray = ARRAY[1..1000] of INTEGER;
      numPtr = ^numArray;
   VAR
      nums : numArray;
      dot : Rect;
      x, y, j : INTEGER;

   BEGIN
      nums := numPtr(theRgn^)^;
```

Figure 4-26. The Region Screen



```
        IF nums[1] > 10 THEN
           BEGIN
              j := 6;
              REPEAT
                 y := nums[j];
                 j := succ(j);
                 REPEAT
                    x := nums[j];
                    j := succ(j);
                    SetRect(dot, x-2, y-2, x+2, y+2);
                    PaintOval(dot);
                 UNTIL nums[j] = 32767;
                 j := succ(j);
              UNTIL nums[j] = 32767;
           END; {if}
     END;

BEGIN {main program}
     SetUpSys;

     RgnA := NewRgn;
     SetRectRgn(RgnA, 100, 100, 200, 300);

     RgnB := NewRgn;
     SetRectRgn(RgnB, 100, 200, 300, 300);

     UnionRgn(RgnA, RgnB, RgnA);
     FillRgn(RgnA, ltGray);

     ShowCorners(RgnA);
     ClickButton;
END.
```

Data Structures

The USES section is the same as in previous examples.

The VAR section declares two region handles: RgnA and RgnB. They are handles, not the actual data. They provide access to the data.

Procedures

The first two procedures, "ClickButton" and "SetUpSys", are the same as in previous programs. The third procedure, "ShowCorners", is new. It displays the corner points of a region. It is a general-purpose routine that can display the corners of any region. Use it freely to peek at other regions as you learn to construct them. You will be surprised at how naturally even curved objects such as ovals can be represented by corners; of course,

it takes a lot of corners. That is why there are a thousand entries in the number array "Nums" in this procedure.

The "ShowCorners" procedure expects one parameter of type *rgnHandle* (that is, a *region handle*). Its TYPE section defines two types: "numArray", which is an array of a thousand integers, and "NumPtr", which is a pointer to an object of type numArray. We use these types to access the integers stored in our regions.

In the VAR section of this procedure, the variable "nums", of type "numArray"; "dot", which is a rectangle; and integers x, y, and j are declared.

In the main part of "ShowCorners", the assignment statement:

```
nums :=numPtr(theRgn^)^;
```

grabs the data for the region from where it is stored dynamically and copies it into static storage, where its internal structure can be carefully examined. This statement is rather tricky because it accomplishes something that Pascal normally tries to prevent, namely, accessing one type of data according to the rules for another type. In our case, the data is originally stored as a region, whereas we wish to access it as an array of numbers. Further complications are that the original structure is of variable length and position in memory and is accessed indirectly through a handle ("theRgn"), whereas we wish to access it directly as an array ("nums") of fixed size and position in memory.

Let's see how this statement works, examining it step by step starting with region handle "theRgn". We begin by suffixing a caret to this handle to obtain "theRgn^", a pointer to the region. Next we apply the type identifier "numPtr" like a function to convert this region pointer to an expression of type "numPtr", which normally points to integer arrays of type "numArray".

The use of this type identifier is an example of *type coercion*, a concept we studied in Chapter 3. In this case, we perform the type coercion between the pointers because the data types have different sizes and thus cannot be coerced from one to the other.

The last step is to suffix another caret to obtain an expression of type "numArray", which refers to the data itself as an array of integers. Assigning this expression to the variable "nums" makes the transfer.

The above assignment statement moves the data in the entire structure to "nums". We need to move the data because the region is a dynamic structure and can be located at different places in memory as the program is executing, whereas "nums" is a static structure whose data remains in the same location throughout the execution of the "ShowCorners" proce-

dure. Since we need the data throughout this procedure, we need to secure it at the beginning. (We found this step to be absolutely necessary to make this particular procedure work in a consistent manner.)

The final part of the "ShowCorners" procedure is conditional. That is, if the region is specified by its rgnBBox, with no additional data (nums[1] is equal to 10), then we should not try to decode those data.

Now let's look at the actual code to display the corner points. The corner data begin at the sixth integer of the region data, so we set the index "j" equal to 6. Next we enter an "outer" REPEAT loop, which grabs the data for one horizontal line of data points. First, we get the y coordinate value, then enter an "inner" REPEAT loop to get the successive x coordinates. For each x coordinate, we draw a circle around the indicated corner point, using the x coordinate and the y coordinate picked up earlier. Notice that each time we pick up a coordinate, we advance our index "j". This imitates the auto-incremented addressing mode that the MC68000 processor would use if programmed in assembly language.

To draw the circle around the corner point, we define the rectangle using the SetRect procedure, then invoke the PaintOval procedure to fill a circle of that size. The "SetRect" procedure assigns specified coordinates to a specified rectangle. It expects five parameters: a rectangle, and four integers that will become its coordinates. They appear in the following order: left, top, right, bottom. We used a circle rather than a single pixel because single pixels are hard to see and slightly offset from the points that they correspond to. (See the earlier discussion on the relationship between pixels and coordinates.)

The inner REPEAT loop ends with an UNTIL statement that checks for the value of 32767. After this loop, we advance our index "j" to look for the next y coordinate. The UNTIL statement at the end of the outer loop checks for y values of 32767, which indicate the end of the entire data structure.

The Main Program

Next comes the main part of the program. We call "SetUpSys" to perform the initialization.

The remainder of the main program defines the L-shaped region and displays it and its corners. The region is defined by the union of two rectangular regions. The first region is the vertical part of the "L", defined by the statements:

```
RgnA := NewRgn;
SetRectRgn(RgnA, 100, 100, 200, 300);
```

The second region is the horizontal part, defined by the statements:

```
RgnB := NewRgn;
SetRectRgn(RgnB, 100, 200, 300, 300);
```

In each case, the region is created with QuickDraw's NewRgn function, which gives the region handle to the newly created region; then the SetRectRgn routine defines it as a 10-byte rectangular region with the specified corner coordinates. The L-shaped rectangle replaces the first region. The routine UnionRgn is used as follows:

```
UnionRgn (RgnA, RgnB, RgnA);
```

It computes the union of the first two arguments and places the results in the third. Notice that the third argument (destination) is the same as the first argument (one of the sources). This poses no problem, since QuickDraw always computes the destination in a separate location, then adjusts the pointers once the calculation is complete. Remember that these are region handles, not the regions themselves nor pointers directly to them. This third level of reference by handles frees QuickDraw to work dynamically and yet allows the programmer to reference the regions at any time.

Next, we call "FillRgn" to fill the region with light gray. This routine expects two parameters: a region handle and a pattern. In our program, we pass the handle "RgnA" in the first parameter and the pattern "ltGray" in the second parameter. The pattern "ltGray" is a default variable initialized by "InitGraf".

We then call our "ShowCorners" procedure to display the corner points:

```
ShowCorners(RgnA);
```

Again, we follow the same policy by referring to the region by its handle.

Finally, we call "ClickButton" to wait for the button click.

Scan Conversion of Regions

Drawing a region involves scanning the region data and drawing the shape line by line on the screen. This is called scan conversion because it converts data into a series of scan lines on the screen. It is possible to write a Pascal procedure to do this. However, QuickDraw already does this so well that we won't try to beat it. You might want to try, or you might want to use a debugger to check the machine code in ROM. The

code for painting a region starts at location $408DA8 in the Macintosh ROM. It is quite complex because it must check myriad details before it performs its own work. You can be happy that Apple took care of this process for you.

Positioning and Sizing GrafPorts

Let's return to grafPorts. Recall that a grafPort is a Pascal record structure containing all current settings of the drawing parameters for your program. In this section, we explore those parameters that control the active drawing area associated with a grafPort.

Each grafPort has several fields that determine its position and size. We see here how these fields specify a local coordinate system for each grafPort.

PortBits

We begin with the second field of a grafPort, a BitMap called ".portBits". It defines the screen from the point of view of the grafPort. The first field, ".baseAddr", gives the base address of the video RAM; the second field, ".rowBytes", gives the number of bytes per row for the video mapping; and the last field of a BitMap is a rectangle called ".bounds".

Initially, portBits is set by "OpenPort" to screenBits, a default variable set by "InitGraf" to the following values:

```
BaseAddr =    $1A700 ($7A700 on the 512K Mac)
RowBytes =        64
Bounds.top =       0
Bounds.left =      0
Bounds.bottom =  342
Bounds.right  =  512
```

Local Coordinates

In this section, we see how the top left corner of the portBits ".bounds" rectangle specifies a "local" coordinate system for each grafPort.

Keep in mind that all coordinate variables in a grafPort are expressed in terms of the local coordinate system for that grafPort. This is the key to understanding the peculiarities of how a grafPort stores information needed to convert between its local and the screen's global coordinates.

Earlier, we described the coordinate system on the screen. This coordinate system is called the *global* coordinate system for the screen. It

originates in the upper left corner of the screen. Coordinate positions in this system correspond directly to pixel positions; that is, the point with coordinates (i,j) is at the upper left corner of pixel [i,j] for each pixel on the screen.

Each grafPort has its own *local* coordinate system in which all its coordinate values are expressed. This local coordinate system is related to the screen's global coordinate system through a simple translation:

```
Ptlocal := Ptglobal + delta
```

where "Ptlocal" is the local coordinates of a point, "Ptglobal" is its global coordinates, and "delta", with the structure of a point, is the "translation vector". In terms of individual horizontal and vertical coordinates, this becomes:

```
Ptlocal.h := Ptglobal.h + delta.h
Ptlocal.v := Ptglobal.v + delta.v
```

You can also solve for global coordinates in terms of local coordinates by the translation:

```
Ptglobal := Ptlocal - delta
```

or

```
Ptglobal.h := Ptlocal.h - delta.h
Ptglobal.v := Ptlocal.v - delta.v
```

Because only a translation is involved, the global and all local coordinate systems have the same scale and orientation, differing only by the position of their origins (see Figure 4-27).

How PortBits Controls Local Coordinates

A grafPort's portBits field holds the relationship between the grafPort's local coordinate system and the screen's global system. However, the information is stored in a cleverly convoluted manner by providing the local coordinates of certain global landmarks.

In particular, the portBits ".bounds" rectangle gives the limits of the video screen in local coordinates. This means that "portBits.Bounds.topleft" contains the local coordinates for the top left point

of the screen, and "portBits.Bounds.botright" contains the local coordinates for the bottom right corner of the screen. This is more than enough information to completely determine the relationship between local and global coordinates. In fact, we need only one corner point.

The top left corner of the screen is the key. It forms a global reference point we can use to good advantage. Since it is the origin of the global coordinate system, its global coordinates are:

```
(0, 0)
```

Since it is the top left corner of "portBits.Bounds", its local coordinates are:

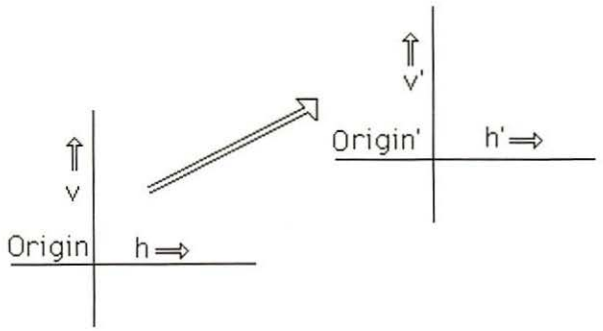
```
PortBits.Bounds.topleft
```

This forces the "delta" in our formulas to be equal to "portBits.Bounds.topleft", thus completely determining these formulas:

```
Ptlocal = Ptglobal + portBits.Bounds.topleft

Ptglobal = Ptlocal - portBits.Bounds.topleft
```

QuickDraw has routines that make these conversions. GlobaltoLocal uses the first formula to convert from global to local coordinates, and LocaltoGlobal uses the second formula to convert from local to global coordinates.

Figure 4-27. Translations



PortRect

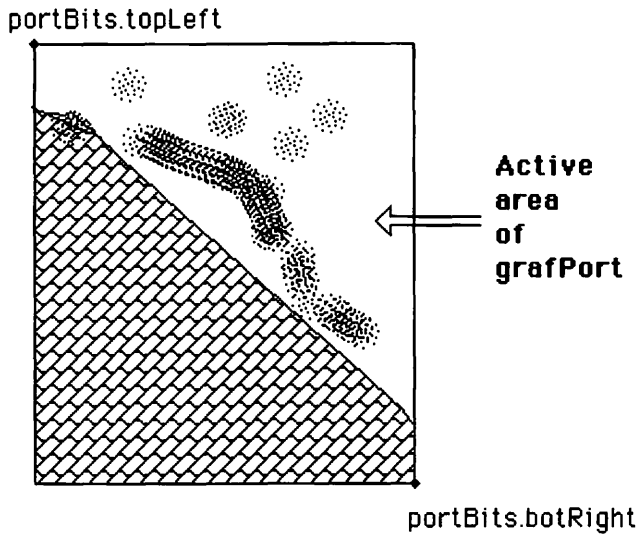
The *portRect* is of type "Rect" and forms the third field of a grafPort. It specifies the *active area* of the grafPort's coordinate system (see Figure 4-28). This is the rectangular area of the screen where your pictures appear. Limiting pictures to rectangular areas of the screen is the first step having *windows*. In Chapter 6, we see how every window is associated with a grafPort whose ".portBits" field defines its *active* or *display* area.

An interesting complication arises when the portRect is specified. Like all coordinates in a grafPort, the coordinates of portRect are local. Life would be easier if global coordinates were used. However, the resulting complications are handled mostly by QuickDraw's port-moving routines. Only when we try to understand exactly what is going on internally does the fun really begin. We look at four QuickDraw port-moving routines, what they do when called, and how they work internally.

QuickDraw Port-Moving and Sizing Routines

Four QuickDraw routines affect the portRect: "MovePortTo" moves the active area relative to the screen, "SetOrigin" moves the origin relative to

Figure 4-28. The Active Area of the GrafPort



the active area, and "PortSize" and "ClipRegion" control the active area's width and height.

These routines are normally called only by the Window Manager routines to position and size a window on the screen, as discussed in Chapter 6. However, our examples show how these routines work in the absence of windows. You might say that our understanding is at the pre-window stage right now, just beginning to explore the QuickDraw tools that make windows work.

MovePortTo Routine

Let's start with "MovePortTo". It moves the active area on the screen. More precisely, if x and y are integers, then the Pascal statement:

```
MovePortTo(x,y)
```

moves the top left corner of the active area to the point whose global coordinates are (x,y). Although its *global* coordinates change, its *local* coordinates are not changed by this command.

Internally, the routine simply computes the following "translation vector":

```
delta = portRect.topleft - portBits.Bounds.topleft - (x,y)
```

and adds this to the corner points of portBits.Bounds according to the following formulas (see Figure 4-28):

```
portBits.Bounds.topleft  := portBits.Bounds.topleft  + delta
portBits.Bounds.botright := portBits.Bounds.botright + delta
```

SetOrigin Routine

Next let's look at "SetOrigin". If x and y are integers, then the Pascal statement:

```
SetOrigin(x, y)
```

shifts the grafPort's local coordinate system so that the *local* coordinates of the top left corner point of portRect become (x,y), but its global coordinates are unchanged. That is, the active area does not change its position or size on the screen, but the local coordinates change, moving the local origin (see Figure 4-29).

Note: this is the opposite of the previous routine, which changes the global coordinates but not the local coordinates of this local reference point.

Internally, this routine computes the translation vector:

```
delta := (x,y) - portRect.topleft
```

and adds it to the corner points of portRect and portBits.Bounds, using the following formulas:

```
portBits.Bounds.topleft  := portBits.Bounds.topleft  + delta
portBits.Bounds.botright := portBits.Bounds.topright + delta
portRect.topleft         := portRect.topleft         + delta
portRect.botright        := portRect.topright        + delta
```

The PortSize Routine

The third QuickDraw routine is "PortSize". If x and y are integers, then the command:

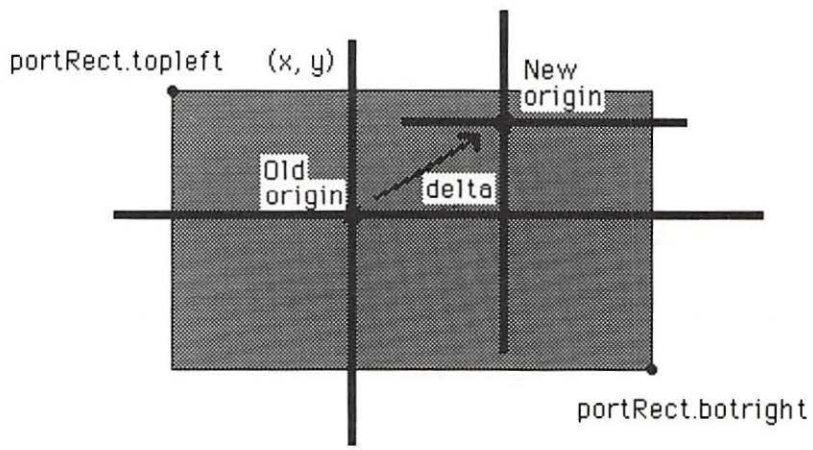
```
PortSize(x,y)
```

changes portRect so that its width is x and its height is y.

Internally, this routine recomputes portRect.botright with the formula:

```
portRect.botright := portRect.topleft + (x,y)
```

Figure 4-29. Setting the Local Origin



Visibility and Clipping

In this section, we discuss how visibility and clipping are implemented in QuickDraw by the ".visRgn" and the ".clipRgn" fields of the grafPort. We also present three routines to help the programmer manage clipping.

Visibility

Visibility relates to the way images overlap (see Figure 4-30). This is especially important in managing multiple overlapping windows, discussed in detail in Chapters 6 and 7. However, we preview it here in our "pre-window" stage of understanding.

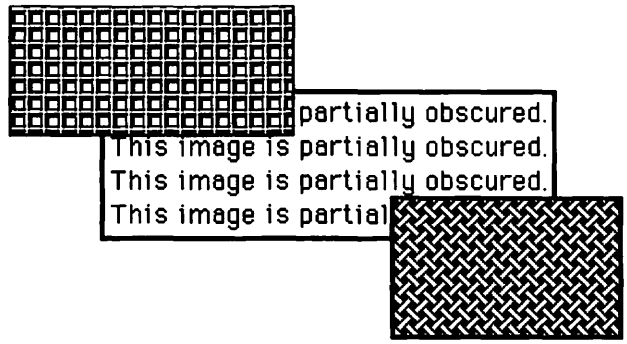
As we will see in Chapter 6, each window has its own grafPort with its own *visibility region*. The grafPort field ".visRgn" is a handle to this region (see Figure 4-31). When a window changes position relative to the screen and to other windows, the Window Manager uses this region to tell QuickDraw which parts of the window need redrawing during a "window updating" process because they are now visible. In Chapters 6 and 7 you will see (and hear) examples.

Clipping

Clipping relates to the way an image is "mounted" on a grafPort (see Figure 4-32). This in turn relates to the way an image is mounted on a window.

Each grafPort has a region called its *clipping region*. The grafPort field ".clipRgn" is a handle to this region (see Figure 4-33). Any parts of shapes falling outside this region are "clipped" (not drawn). As we see

Figure 4-30. Visibility



later, the programmer can control this clipping region to create images that fall within the desired (active) area of the screen.

Relation between Visibility and Clipping

Visibility and clipping are treated separately by the Macintosh. One is controlled by the Window Manager, the other is controlled by the programmer. However, QuickDraw uses both, drawing only those pixels in the intersection of both regions.

Because QuickDraw automatically draws only what is in both regions, the programmer can freely draw images, not worrying about whether pixels fall "within the window" or are currently hidden by another window.

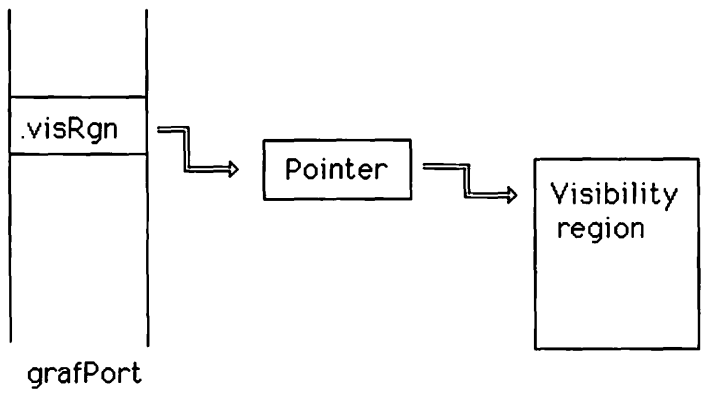
Two different regions simplify the management of windows, allowing the Window Manager and the programmer separate data structures that do not interfere with one another.

Clipping Routines

Now let's see how we can directly control the clipping region. The QuickDraw procedure *SetClip* sets the clipping region equal to a specified region. It expects a single parameter, which is a region handle that leads to a valid region.

Before you use this routine, your region handle must be associated with an actual region. Use the QuickDraw routine "NewRgn" to allot room for regions and routines such as "SetRectRgn" and "UnionRgn" to define their shapes.

Figure 4-31. The Visibility Region



The QuickDraw routine "GetClip" returns the current clipping region. The two routines "GetClip" and "SetClip" perform opposite jobs and can operate in conjunction to manipulate the current clipping region.

The procedure "GetClip" expects a single parameter, which is a region handle associated with a valid region. After this routine is called, this region handle leads to a copy of the current clipping region. The region handle does not change: it is not a "VAR" parameter, so it still

Figure 4-32. Clipping

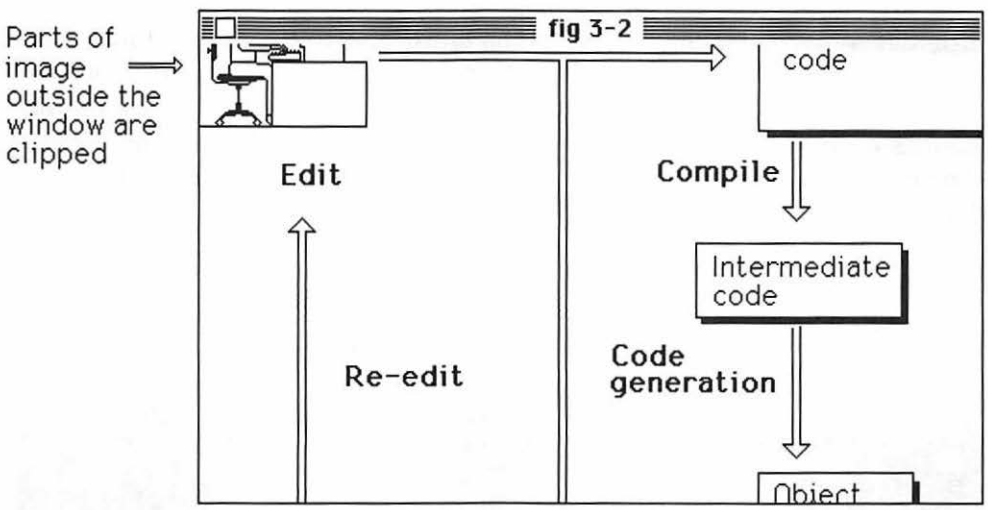
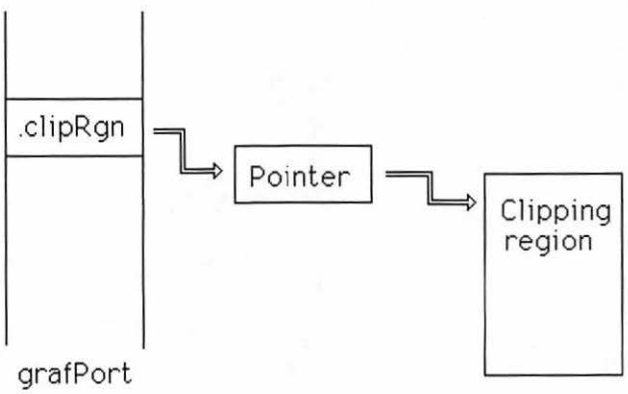


Figure 4-33. The Clipping Region



points to the same region pointer. However, this region pointer normally points to a new place on the heap where the copy of the current clipping region is stored. Note that this copy is disassociated from the current clipping region: it is stored in a different place from the current clipping region and uses different pointers and handles.

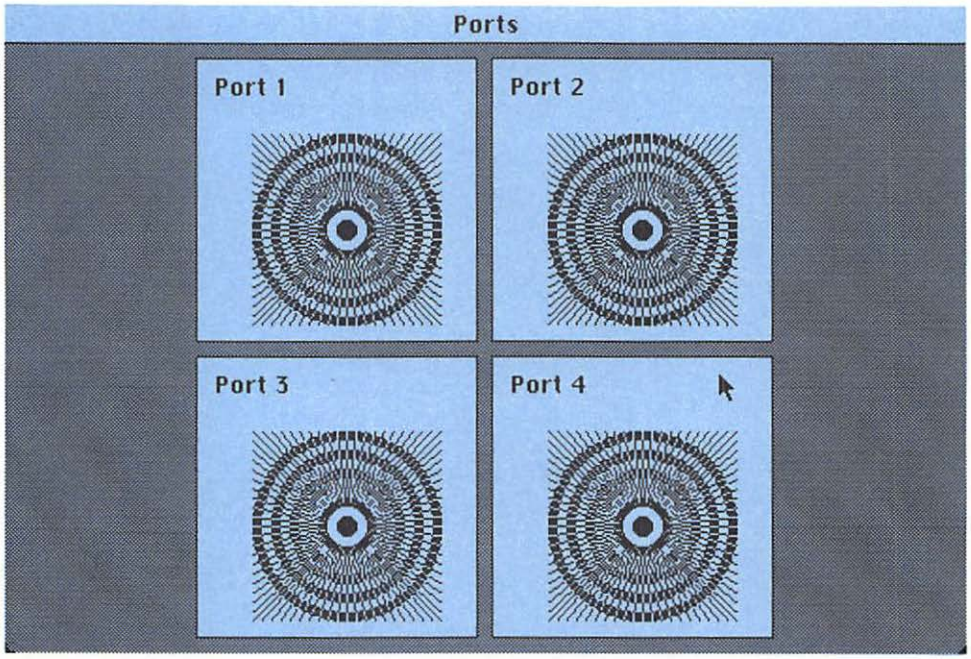
The Ports QuickDraw Example

Here is a Pascal program called "Ports" that illustrates how grafPorts can be moved and sized. It even demonstrates how to manage several grafPorts at once. It uses clipping to restrict each image to its own grafPort.

The program opens with a blank screen filled with gray and a white title bar. When the cursor turns to the familiar arrow, you proceed.

Click the mouse and you see four rectangular areas appear. They are labeled "port 1" through "port 4". Another click of the mouse button causes a design to be drawn in all four ports at once (see Figure 4-34). The next click causes the entire display to invert. A final click terminates the program.

Figure 4-34. The Ports Program



Here is the program:

```
PROGRAM Ports;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf;

   VAR
      savePort : GrafPtr;
      Port : array[1..4] of GrafPtr;

   PROCEDURE ClickButton;
      BEGIN
         WHILE Button DO;
         WHILE NOT Button DO;
         WHILE Button DO;
      END;

   PROCEDURE SetUpSys;
      BEGIN
         InitGraf(@thePort); {Initialize default variables}
         InitFonts;          {Initialize Font Manager }

         NEW(thePort);       {Set up grafPort}
         OpenPort(thePort);
         InitCursor;
      END;

   PROCEDURE OPort(J, x, y : INTEGER; title : Str255);
      BEGIN
         New(Port[J]);
         OpenPort(Port[J]);

         PortSize(150,150);
         ClipRect(thePort^.portRect);
         MovePortTo(x,y);

         EraseRect(thePort^.portBits.Bounds);
         FrameRect(thePort^.portRect);

         MoveTo(10,20);
         DrawString(title);
      END;
```

```
PROCEDURE DrawPorts;
   BEGIN
      GetPort(savePort);
      OPort(1, 102,  27, 'Port 1');
      OPort(2, 259,  27, 'Port 2');
      OPort(3, 102, 184, 'Port 3');
      OPort(4, 259, 184, 'Port 4');
   END;

PROCEDURE DrawDesigns;
   VAR
      I,J : Integer;
      R : Rect;

   BEGIN
      FOR I := 0 to 20 DO
         FOR J := 1 to 4 DO
            BEGIN
               SetPort(Port[J]);
               MoveTo(       30, 40 + 5*i);
               LineTo(      130,140 - 5*i);
               MoveTo( 30 + 5*i,       40);
               LineTo(130 - 5*i,      140);
            END;

      FOR I := 1 to 10 DO
         BEGIN
            SetRect(R, 80 - 5*i, 90 - 5*i, 81 + 5*i, 91 + 5*i);
            FOR J := 1 to 4 DO
               BEGIN
                  SetPort(Port[J]);
                  InvertOval(R);
               END;
         END;
    END;

PROCEDURE InvertScreen;
   BEGIN
      SetPort(savePort);
      InvertRect(thePort^.portBits.Bounds);
   END;

BEGIN {main program}
   SetUpSys;
   ClickButton;

   DrawPorts;
   ClickButton;
```

```
    DrawDesigns;
    ClickButton;

    InvertScreen;
    ClickButton;

END.
```

Data Structures

The "Ports" program has the standard USES section. Its VAR section declares two global variables: "savePort", which is a grafPtr, and "Port", which is an array of four grafPtrs.

Procedures

The "Ports" program has a number of procedures, including "ClickButton" and "SetUpSys", that have already appeared. However, "Ports" has four new routines: "Oport", "DrawPorts", "DrawDesigns", and "InvertScreen".

Setting Up a Port

The procedure "OPort" sets up a specific grafPort of our array of grafPorts.

The "OPort" procedure has four parameters: three integers — "J", "x", and "y" — and a string, "title". The first integer selects one of the four grafPorts. The second and third integers specify the position of its upper left corner. The string "title" specifies a title for the grafPort.

The procedure begins by calling "New" to allot space for the grafPort "Port[J]", then calls "OpenPort" to initialize it.

Next, we call "PortSize" and "ClipRect" to specify the size of this grafPort and "MovePortTo" to specify its position. These QuickDraw routines operate on certain rectangles and regions associated with the current grafPort as described above.

We call "EraseRect" to erase it and "FrameRect" to draw a border around it. Then we call "MoveTo" to position the title and "DrawString" to draw the title.

Drawing the Ports

The procedure "DrawPorts" draws all four ports. It begins by calling the QuickDraw routine "GetPort" to save the current grafPort in our variable "savePort". Next, it calls our "OPort" procedure four times, once for each grafPort.

Drawing Designs

The procedure "DrawDesigns" provides the fireworks. It draws the same figure on all four drawing areas at once by rapidly switching among the four different grafPorts.

The "DrawDesigns" procedure has three local variables: two integers and a rectangle. The two integers are indices to FOR loops within the procedure, and the rectangle draws the circles in the design.

The procedure contains two double FOR loops. Within each FOR loop, another FOR loop switches among the grafPorts, drawing the same tiny part of the figure on each grafPort. Even though the drawing is not simultaneous in each grafPort, the effect is the same: much like a time-sharing computer that seems to simultaneously serve several users by rapidly switching among them.

Let's look at the figure carefully. As noted earlier, the figure is drawn in two stages, using two different FOR loops. In the first FOR loop, we draw a set of radial lines through the center of a square. In the second stage, we invert a sequence of increasingly larger circles, creating a series of square rings that cross the radial lines.

Inverting the Screen

The procedure "InvertScreen" inverts the entire screen. It begins by calling "SetPort" to restore the current grafPort to what was saved by the "savePort" grafPtr. In this program, we called "GetPort" earlier to place the original grafPort in this special save grafPtr.

Next, it calls "InvertRect" to invert the rectangle "thePort^.portBits.Bounds". This is the bounds rectangle for the bitMap "portBits", which is a field of the current grafPort.

The Main Program

The main part of the program calls our routines, one after another, each followed by a call to "ClickButton" to pause for the user to hit the mouse button.

It calls "DrawPorts" to draw and label the four grafPorts. It calls "DrawDesigns" to draw the design on all four ports. Finally, it calls "InvertScreen" to invert the entire display.

Summary

In this chapter, we have introduced some of the basic concepts of QuickDraw: hardware, such as memory-mapped video; mathematics, such as coordinates and coordinate systems; graphics, such as clipping and visibility; and QuickDraw data structures, such as pattern, cursor, point, Rect, BitMap, region, and grafPort.

We have presented six short example programs that illustrate these structures.

In future chapters, we introduce more QuickDraw concepts, data structures, and routines as we need them.

The following ROM routines are covered in this chapter:

EM-Button

QD-InitGraf

QD-OpenPort

QD-InitCursor

FM-InitFonts

QD-StuffHex

QD-FillRect

QD-SetPt

QD-SetCursor

QD-HideCursor

QD-EraseRect

TU-BitSet

EM-GetMouse

QD-FrameRect

QD-PaintRect

QD-InvertRect

QD-SetRect

QD-SetRectRgn

QD-UnionRgn

QD-FillRgn

QD-PortSize

QD-ClipRect

QD-MovePortTo

QD-MoveTo
QD-DrawString
QD-GetPort
QD-SetPort
QD-LineTo
QD-InvertOval

5

Introduction to Events

This chapter covers the following new concepts:

- **Events**
- **The Event Manager**
- **The Event Queue**
- **Vertical Retrace Manager**
- **Event Records**
- **Initialization of the Event Manager**
- **Addressing the Video RAM**
- **Accessing Events**
- **Keyboard and Mouse Events**

In this chapter, we introduce *events*, a *dynamic* method of communication between the Operating System and an applications program. Events are the cornerstone of Macintosh's *interactive* programming capabilities, providing a way to organize input from the user into a form that can easily be handled by an applications program.

Events are the official channel for transmitting user input to an applications program and a way to schedule other activities, such as updating the screen, that require coordination between the applications program and the Operating System.

For the Macintosh, an *event* is a *record* (stored in a Pascal record structure) of a specific action, such as pressing and releasing a key on the

keyboard or the button on the mouse, inserting a disk in the disk drive, or a window changing position or coming to the front of the display screen. However, not all user actions generate an event. For example, moving the mouse across your desk without pressing the button is not an event.

In less advanced computers, input from the user comes from the keyboard only. The applications program checks for or waits for single keystrokes or entire strings of text. Output consists of strings of text displayed line by line on the screen. Other parts of the system, such as the disk, are tended to separately.

With the Macintosh, the situation is more complex, since input comes from both the keyboard and the mouse and output is handled through multiple overlapping windows. Each subsystem requires special attention. However, all Macintosh input and output can be reduced to a series of *events*, each requiring specific *action* from the applications program.

The Macintosh's *Event Manager* monitors each event, placing a Pascal record structure containing vital information in a waiting line called a *queue*. The applications program can request these event records from the queue, one at a time, as it can handle them. The applications program can distinguish among the different types of events (key down, key up, mouse button down, mouse button up, disk insertion, screen update) by examining a field called ".what". It can also examine other fields, such as ".where" to see where the mouse was and ".when" to see the time that the event occurred. Still other fields tell such things as key codes for keyboard events and identification numbers for screen update events.

The event approach is an advantage: it sorts events for the applications program, presenting them in a uniform manner, allowing the applications program to concentrate on managing the system at a much higher level. This approach is more efficient. The Operating System can quickly handle each event, place it on the queue, then go to the next event or other task.

In this chapter, we introduce the Macintosh's *Event Manager* and explain its relation to the rest of the system. We describe *interrupts* that generate events and event information at the lowest levels, the *record* structure that stores events, and the *linked list* structure used by the *event queue*.

We also present an example program that illustrates how events appear to the user and how they are programmed. This straightforward model allows you to create your own interactive, event-driven programs. The model demonstrates keyboard events and mouse events, forming a foundation for programming other types of events. In later chapters, we study how the Event Manager interacts with the Window Manager to control the Macintosh's multiple overlapping windows.

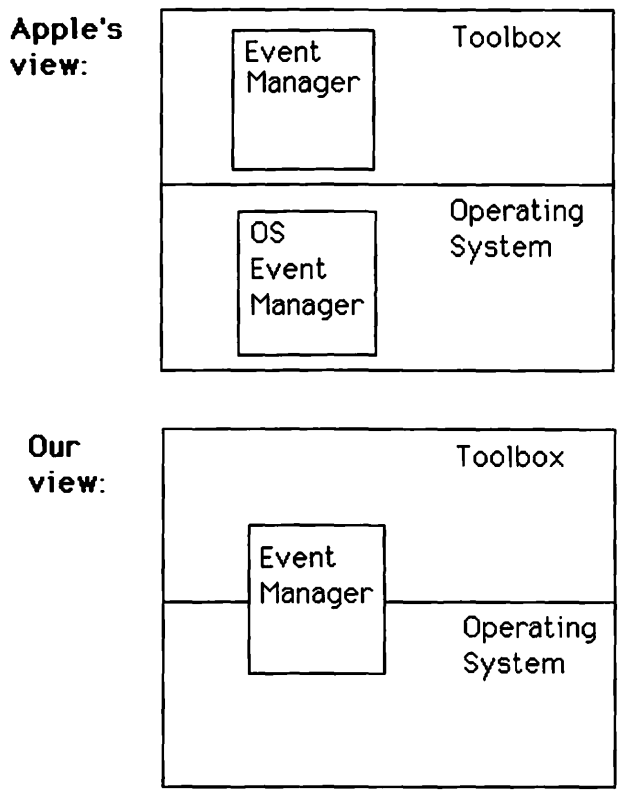
The Event Manager

As we have seen, events are handled by the *Event Manager*, which is a set of routines and data structures in the Macintosh's memory.

In its first manuals, Apple refers to two event managers, one in the Operating System, one in the Toolbox (see Figure 5-1). But such categorizations are somewhat arbitrary. In particular, the distinction between Operating System and Toolbox is vague; in fact, Apple has moved certain Event Manager data structures from one category to the other as the software for the Macintosh has matured.

We differ from Apple in that we consider there to be only one Event Manager — partly within the Operating System, partly within the Toolbox. This makes the distinction between Operating System and Toolbox less critical, and rearrangements of the two parts of the system seem more natural.

Figure 5-1. The Event Manager(s)



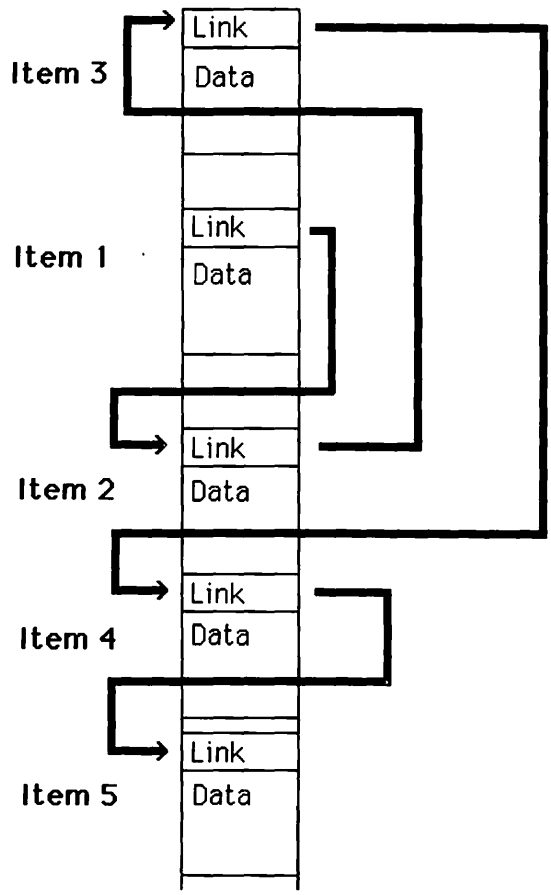
The Operating System Part

The Operating System part of the Event Manager (the OS Event Manager, for short) forms the lower levels of the Event Manager. This contains routines to manage the *event queue*. We use some of these routines in our program, so it is important to understand some details of this structure.

Linked Lists

The Macintosh maintains a list or *queue* of pending events in a *linked list*. This is a *data structure* consisting of items in which each item contains a pointer to (holds the address of) the next (see Figure 5-2). Thus,

Figure 5-2. A Linked List



in a linked list the items themselves supply the information needed to hold the list together, and yet these items need not be next to each other. In fact, the items can be scattered in any order throughout memory, with the pointers forming a thread that ties them together in the intended order.

Linked lists are particularly useful because of the ease with which they can be manipulated. For example, operations such as appending new items to the end, inserting new items in the middle, and deleting old items from anywhere in the list can be done quickly and efficiently, simply by placing new values in the pointers. By contrast, performing such operations on ordinary lists that do not have this linking structure requires their items to be moved around in memory at considerable loss of performance.

Linked lists are advantageous when each item contains a moderate to large amount of data. However, they do require extra memory to hold the pointers and are therefore less desirable when each item is small. In fact, if we tried to organize lists of individual bytes as linked lists, we would spend much more memory on the linking than on the data. In the case of Macintosh events, however, there is enough data in each item to make a linked list an appropriate choice. In addition, the ability to place information in queues using this linking greatly increases the efficiency of the system. It takes very little time to add a new item to a queue, and once it's done, the system is free to go on to other matters.

The routines in the Operating System part of the Event Manager can add, fetch, remove, and check events on this queue. We will use some of these routines in our example program. Note that the Macintosh's Operating System uses linked lists to manage other mechanisms, such as disk volumes and I/O service requests.

Calls from Lower Levels

The *Vertical Retrace Manager* maintains a number of *background* tasks for the Macintosh. These are jobs that operate independently of and at the same time as the applications program. For the Macintosh, these include updating the mouse cursor on the screen, updating the time and date, and monitoring the mouse button and disk insertion.

The *Vertical Retrace Manager* is driven by an interrupt called the *Vertical Retrace Interrupt*, which is generated by the video hardware each time the video signal is blanked between scans of the screen. An *interrupt* is a hardware signal that causes the processor to stop, process the activity, then return to its previous task.

Calling the Vertical Retrace Manager in this way increments a system variable called the *tickCount* and checks several functions, including the

system stack and the mouse position. The Vertical Retrace Manager also checks the mouse button every other time and the disk system every thirty times. It updates the cursor position if the mouse has moved (mouse movement itself is handled by another interrupt). If the mouse button has changed, the Vertical Retrace Manager calls the Event Manager to place a mouse event on the queue. Similarly, it places a disk event on the queue, if appropriate.

Calls from Higher Levels

At higher levels, the OS Event Manager routines may be called by the routines in the Toolbox part of the Event Manager or by an applications program.

The Toolbox Part

The Toolbox contains higher-level routines that interface the event queue to the user. We explore these higher-level routines in the remainder of this chapter.

Example Program

The example program demonstrates how to program events generated by the keyboard and the mouse button. This introduces the basic concepts needed to program other events. In Chapter 6, we will see how to handle screen updating events generated by the Window Manager.

The "Events" program graphically demonstrates keyboard and mouse activity. It displays the results of this activity in several boxes on the screen (see Figure 5-3). The two boxes with square corners show keyboard events, and the three boxes with rounded corners interact with the mouse.

The program itself draws the boxes on the screen and then monitors events and other information from the Event Manager, displaying information in these boxes. For example, if you hit a key on the keyboard, the Events program displays the corresponding character in a box called "Key".

The program also displays the keyboard as a row of bits on the screen in a box called "Keyboard Array". Each bit corresponds to a particular key position. Pressing a key blackens the corresponding bit on the screen.

At the same time, the program monitors mouse activity. The cursor moves around the screen as you move the mouse, and if you press the mouse button while the cursor is in certain boxes, then different things will happen depending on where the mouse is. If you press the button while the cursor is in the box labeled "Mouse Points", you will see points

(mouse droppings) appear within that box. If you press the button while the cursor is in the box labeled "Erase mouse points", the mouse droppings will be erased. Finally, if you press the button while the cursor is in the box labeled "Exit", the program will terminate.

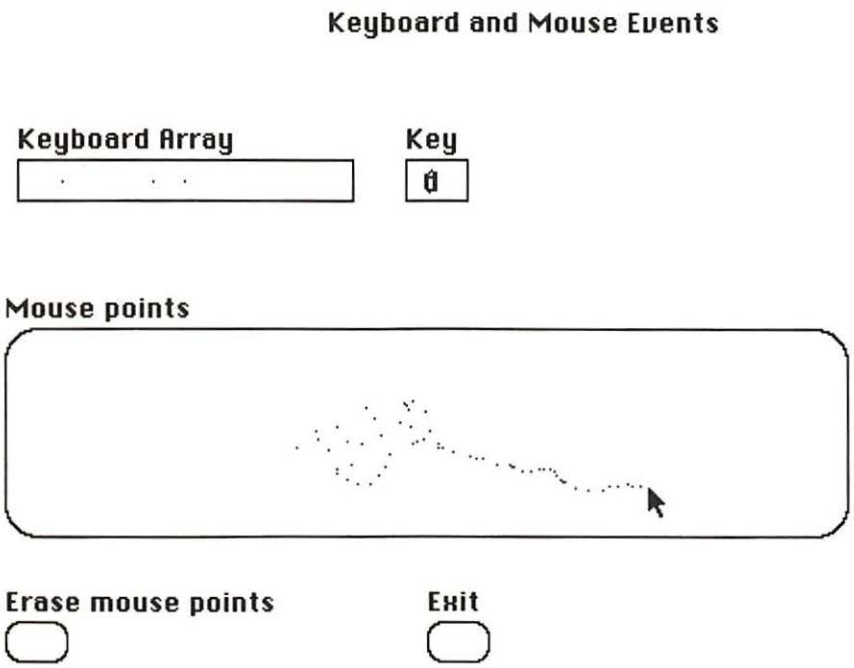
The mouse boxes are implemented by QuickDraw *regions*, a concept introduced in Chapter 4. *Regions* allow us to easily determine when the mouse is in a more complex shape such as a rounded rectangle. QuickDraw has a built-in procedure that can quickly determine whether a particular point is in a particular region.

The mouse boxes in this program are primitive versions of *controls*. In Chapter 6, you will see how the Macintosh's *Control Manager* uses regions to automatically define and handle controls.

Now let's look at the program.

```
PROGRAM Events;
   {$R-}{$X-}
```

Figure 5-3. Screen Layout for Events Program



```
USES
   {$U obj/Memtypes    } Memtypes,
   {$U obj/QuickDraw   } QuickDraw,
   {$U obj/OSIntf      } OSIntf,
   {$U obj/ToolIntf    } ToolIntf;

TYPE
 KeyPtr = ^KeyMap;

VAR
  theEvt : EventRecord;
  done : BOOLEAN;
  theKeys : KeyPtr;
  MPt : Point;
  kbRect, keyRect, mouseRect, eraRect, exitRect : Rect;
  mouseCurv, eraCurv, exitCurv : Point;
  mouseRgn, eraRgn, exitRgn : rgnHandle;

FUNCTION VideoAddr(x, y : INTEGER) : LongInt;
   BEGIN
    VideoAddr := ORD(screenBits.BaseAddr)
                 + x DIV 8
                 + y*screenBits.rowBytes;
   END;

PROCEDURE Setup;
   BEGIN
      InitGraf(@thePort);
      InitFonts;

      NEW(thePort);
      OpenPort(thePort);

      FlushEvents(EveryEvent,0);
      SetEventMask(EveryEvent);

      InitCursor;
      EraseRect(thePort^.portBits.Bounds);
   END;

PROCEDURE MakeTitle (title : Str255);
   {Draw a centered title}
   BEGIN
      WITH thePort^.portRect DO
        MoveTo((left + right - StringWidth(title)) DIV 2, 20);
      DrawString(title);
   END;
```

```
PROCEDURE MakeRect(R : Rect; title : Str255);
   BEGIN
     {Draw and label rectangle}
      EraseRect(R);
      FrameRect(R);
      MoveTo(R.left, R.top-5);
      DrawString(title);
   END;

FUNCTION MakeRRgn(R : Rect; Curv : Point) : RgnHandle;
   VAR
      Rgn : RgnHandle;
   BEGIN
     {Define a rounded rectangular region}
      Rgn := NewRgn;
      OpenRgn;
       FrameRoundRect(R, Curv.h, Curv.v);
      CloseRgn(Rgn);
      MakeRRgn := Rgn;
    END;

PROCEDURE DrawRRgn(Rgn : RgnHandle; title : Str255);
   BEGIN
     {Draw and label rounded rectangular region}
      EraseRgn(Rgn);
      FrameRgn(Rgn);
      WITH Rgn^^.RgnBBox DO MoveTo(left,top-5);
      DrawString(title);
   END;

BEGIN {main program}
   Setup;
   MakeTitle('Keyboard and Mouse Events');

  {Define rectangles and curvatures for regions}
   SetRect(kbRect,16, 80, 176, 100);
   SetRect(keyRect,200, 80, 230, 100);
   SetRect(mouseRect,10, 160, 410, 260);
   SetRect(eraRect,10, 300, 40, 320);
   SetRect(exitRect,210, 300, 240, 320);
   SetPt(mouseCurv, 32, 32);
   SetPt(eraCurv, 16, 16);
   SetPt(exitCurv, 16, 16);

  {Set up display for keyboard array}
   MakeRect(kbRect,'Keyboard Array');
   theKeys := POINTER(VideoAddr(kbRect.left+16,kbRect.top+10));

  {Set up box for key display}
   MakeRect(keyRect, 'Key');
```

```
{Set up MousePoint region}
 mouseRgn := MakeRRgn(mouseRect, mouseCurv);
 DrawRRgn(mouseRgn, 'Mouse points');

{Set up Erase box region}
 eraRgn := MakeRRgn(eraRect, eraCurv);
 DrawRRgn(eraRgn, 'Erase mouse points');

{Set up Exit box region}
 exitRgn := MakeRRgn(exitRect, exitCurv);
 DrawRRgn(exitRgn, 'Exit');

 REPEAT { main loop }
   If GetNextEvent(everyEvent, theEvt) THEN
      CASE theEvt.what OF

         mouseDown: {track the mouse}
            BEGIN
               GetMouse(MPt);

               If PtInRgn(MPt,mouseRgn) THEN BEGIN
                  MoveTo(MPt.h, MPt.v);
                  Line(0,0);
               END;

               If PtInRgn(MPt,eraRgn) THEN
                  DrawRRgn(mouseRgn, 'Mouse points');

               If PtInRgn(MPt,exitRgn) THEN
                 done := true;

            END;

         keyDown: {display key character}
            BEGIN
               MoveTo(keyRect.left+10, keyRect.bottom-5);
               DrawChar(chr(theEvt.message MOD 256));
            END;

         keyUp:  {erase key character}
            MakeRect(keyRect, 'Key');

      END;

      GetKeys(theKeys^);  { Display key map }

   UNTIL done;
END.
```

External Files

The USES section of programs that involve events should invoke the following external files: "MemTypes", "QuickDraw", "OSIntF", and "ToolIntf". The first two files were discussed in Chapter 4.

The third external file, "OSIntf", defines parts of the Event Manager that are in the Operating System. This now includes all of its data structures and entry to some of its lower-level routines.

The fourth file, "ToolIntf", defines the Toolbox part of the Event Manager. These are its higher-level, or user-oriented, routines.

Data Structures

In the TYPE section of the "Events" program, we define a type, "KeyPtr", which points to the variables of the type "KeyMap". The type "KeyMap" is defined in the external files as:

```
KeyMap = ARRAY[0..3] OF LONGINT;
```

This definition has changed since the first appearance of the Macintosh development system. Originally, it was a packed array of 128 Boolean variables! It occupied the same amount of storage, since each Boolean variable was stored in its own bit.

The VAR section declares the global variables for this program.

Event Records

The first global variable is "theEvt", which is an *event record*. This data structure transmits information about the event to the applications program. An *event record* can be decribed by the following Pascal structure:

```
EventRecord = RECORD
                what      : INTEGER;
                message   : LongInt;
                when      : LongInt;
                where     : Point;
                modifiers : INTEGER
              END;
```

This format is somewhat like a recorded telephone message. It contains information such as the time as well as specific information about what happened.

The first field, ".what", contains an integer that specifies the type of event that occurred. There are 16 possibilities, one for each bit position in a 16-bit integer. Later, we will see how all the event types are sometimes combined into one 16-bit integer that forms a "mask" indicating which types are selected or active. Currently, only 14 types are supported, as defined by the following constants statements:

```
nullEvent = 0;
mouseDown = 1;
mouseUp = 2;
keyDown = 3;
keyUp = 4;
autoKey = 5;
updateEvt = 6;
diskEvt = 7;
activateEvt = 8;
driverEvt = 11;
app1Evt = 12;
app2Evt = 13;
app3Evt = 14;
app4Evt = 15;
```

To make your programs more readable, use these identifiers rather than the corresponding raw integer values.

This chapter deals only with the following events: "null", "mouseDown", "keyDown", and "keyUp". A "null" event is reported if no events of the specified types are in the queue. Chapter 6, which introduces *windows*, also discusses events of types "updateEvt" and "activateEvt".

The second field, ".message", is a long integer containing specific information about the event. The exact format depends on the type of event.

For the keyboard events "keyDown", "keyUp", and "autoKey", the lowest byte of the message contains the extended ASCII code for the key, and the next to lowest byte of the message contains its position number on the keyboard matrix (see Figure 5-4).

The message field for mouse events is zero.

The third field, ".when", gives the time that the event occurred, expressed in "ticks" since the system was last started (turned on or rebooted).

The fourth field, ".where", gives the global coordinates of the mouse when the event occurred.

The fifth field, ".modifiers", is an integer (16-bit computer word) containing information about the modifier keys, mouse button, and other relevant data (see Figure 5-5). The modifier keys and button are each represented by a bit in this computer word. Figure 5-5 shows how.

Done Flag

The next variable declared in the example program is a BOOLEAN called "done". It controls the main loop of the program, which terminates when "done" becomes TRUE. Thus, its name aptly describes the role of this variable, making the loop control self-documenting.

KeyPtr

The next variable, "theKeys", is of type "KeyPtr", which was defined in the TYPE section. We have already discussed how this variable helps us to display the keyboard matrix.

Mouse Point

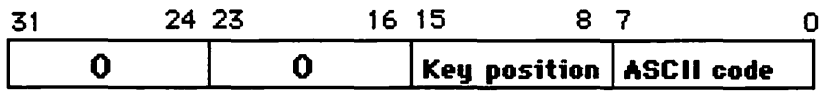
The variable "MPt", of type "Point", keeps track of the position of the mouse.

QuickDraw Variables

The rest of the variables in this program are QuickDraw data structures that define the shapes of the boxes displayed on the screen.

Five rectangles define the sizes of these display boxes: "kbRect", "keyRect", "mouseRect", "eraRect", and "exitRect".

Figure 5-4. Message Field for Keyboard Events



Three points contain information about the roundedness or curvature of the three mouse boxes: "mouseCurv", eraCurv", and "exitCurv" (see Figure 5-6). For rounded rectangles that define the shapes of these boxes, the horizontal components of the points give the oval width, and the vertical components of the points give the oval height.

Figure 5-5. Modifier Field for Events

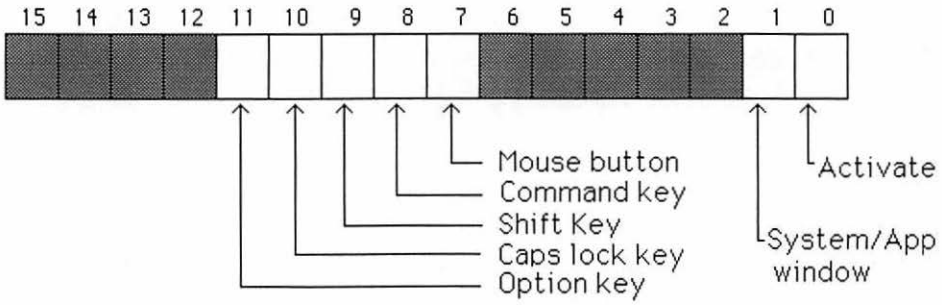
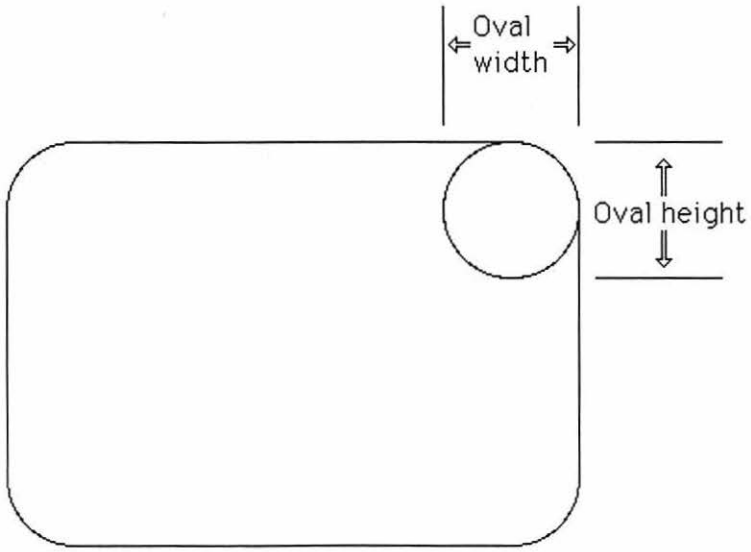


Figure 5-6. Specifying Rounded Rectangles



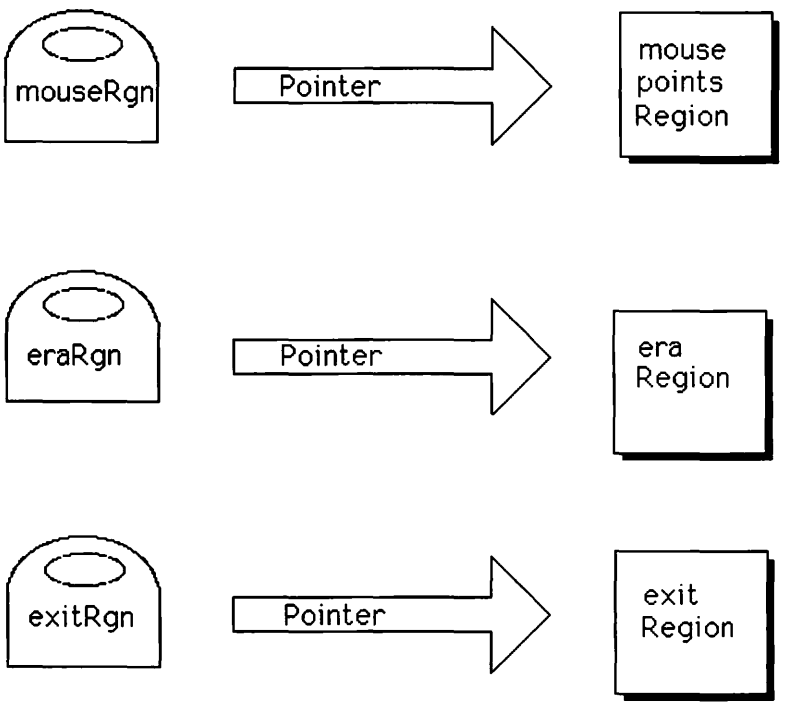
Regions

The last three global variables — "mouseRgn", "eraRgn", and "exitRgn" — are *region handles* that access regions that form the underlying structure of the mouse boxes (see Figure 5-7).

As shown in Chapter 5, a *region* is a fundamental QuickDraw data structure that allows us to define irregularly shaped areas on the screen so they can be quickly and easily displayed and manipulated and can be used for visibility considerations. This chapter discusses how to define irregularly shaped regions and to detect when a point is within an irregular area defined by a region.

The regions in this chapter do not appear very irregular, just rounded rectangles. Yet, even for this simple shape, it would be difficult to write a program completely on your own to detect when a point is within the shape. We see how QuickDraw makes this job trivial for an applications programmer.

Figure 5-7. The Mouse Region Handles



The three mouse region handles provide access to regions, which are dynamic structures stored in the heap. As we have discussed earlier, dynamic structures can grow in size and change position in memory.

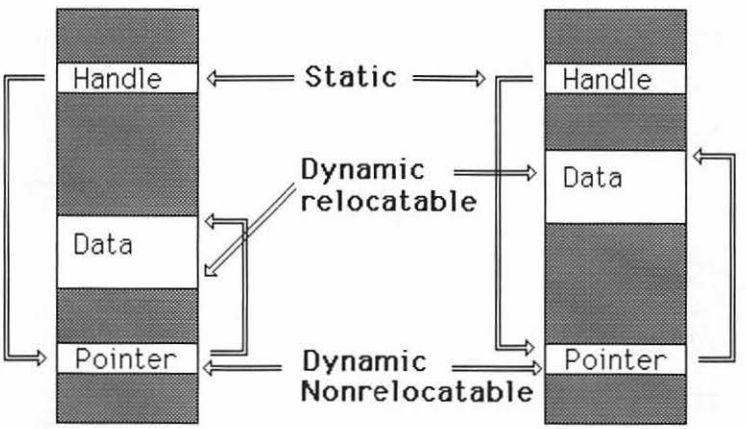
The region handles are static variables stored in the stack area. They will be set to point to dynamic variables called region pointers, which are stored in a special nonrelocatable area of memory in the heap (see Figure 5-8).

These nonrelocatable variables in turn point to the actual region data that are stored in relocatable memory. As the Macintosh's Operating System relocates region data (as the size changes), it automatically updates the region pointers. That is, the Operating System controls both the position of the region data and the values stored in the region pointers, but it does not move the pointers. This way, the region handles never lose track of the regions, even when region data are moved. If the handle pointed directly to the data, then the Operating System would have to update it when data are moved; but the handle is in your program, which is your responsibility, not the responsibility of the Operating System.

Procedures and Functions

Many functions and procedures in this program are general purpose; that is, they can be used in a variety of programs. This is a good general practice. Such general functions help to develop programs with a minimum of effort and apply code developed in one program to many others.

Figure 5-8. Handles and Pointers



Addressing the Screen

The first function is "VideoAddr". It returns the address of the byte in screen memory that contains the bit corresponding to an indicated pixel position. We use this function to place the image of the keyboard matrix on the screen.

The parameters for the "VideoAddr" function are integers "x" and "y". The first variable, "x", gives the horizontal position of the pixel. The second variable, "y", gives its vertical position from the top of the screen.

The "VideoAddr" function uses the default variable "screenBits" to provide its coefficients. This increases the portabilty of the program. Even if the dimensions or location of the screen change, we won't have to refigure the address. The formula for this function is given by the statement:

```
VideoAddr := ORD(screenBits.BaseAddr)
             + x DIV 8
             + y*screenBits.rowBytes;
```

The "BaseAddr" field of screenBits is a pointer to the beginning of video memory. To compute with it, you must use ORD to convert it to a long integer. The function ORD converts pointer values to the numerical value of the corresponding address. To do pointer "arithmetic", you must convert to long integer values. To get the contribution of the x coordinate to the byte address, we must divide it by 8, which is the number of pixels per byte. The "rowBytes" field specifies the number of bytes per row. We use it as the coefficient of the y coordinate.

Initializing the Managers

The procedure "Setup" initializes the managers. We initialize QuickDraw as before with the "InitGraf" routine. However, we also initialize the Font Manager before setting up our grafPort with the "new" and "OpenPort" procedures, as before.

The "Setup" procedure continues by initializing the Event Manager. It calls the "FlushEvents" and the "SetEventMask" routines. The first is a ROM routine, the second is a RAM routine contained in an external file linked to your program; thus, this routine is loaded into memory with your program if it is used in your program. We will explain these routines, but first let's finish the "Setup" routine that called them.

After "SetEventMask", the "Setup" procedure calls "InitCursor" to make the mouse cursor into a standard arrow; then it calls "EraseRect" to erase the entire screen. The specified rectangle to be erased is the

bounds rectangle of the portBits field for the current grafPort. Since the grafPort was just initialized, we can rely on this being the entire screen.

Event Queue

Now let's look closer at "FlushEvents" and "SetEventMask". As described above, the Event Manager keeps track of events by means of its Event Queue, which we will now look at in detail.

The Macintosh's *event queue* begins with a special *header*, containing a status word (an integer with some Boolean variables in bit form) and pointers to the first and last elements of the event queue (see Figure 5-9). This header is called "EventQueue" and is stored in a data area of the Operating System called the "System Communications Area". The event queue itself is stored in the heap.

Each item in the event queue (see Figure 5-9) can be described by the following structure:

```
evQEl = RECORD
     qLink          : ElemPtr;
     qType          : INTEGER;
     eventdata      : EventRecord;
 END;
```

The first field is a pointer to the next item in the queue. This gives the linking structure. The second field gives the queue type (which in this case has a value of four) because these items belong to the event queue, which is queue number four in the system. The third field is a copy of the event record. This particular description is not exactly the same as that currently used by Apple, but it is equivalent and easier to explain. Departing from Apple in this way is not critical, since Apple tends to change such details as its software matures. This has no effect on how applications are programmed.

With this in mind, let's study *FlushEvents*. Basically, it traces through the event queue, removing selected types of events. It has two 16-bit integer parameters called "eventMask" and "stopMask". Both "masks" are really bit patterns whose individual bits select particular types of events (see Figure 5-10). The bit positions are numbered from zero to 15. A bit value of one selects the corresponding event and a bit value of zero deselects it. This explains why there are a maximum of 16 possible types of events.

The following call:

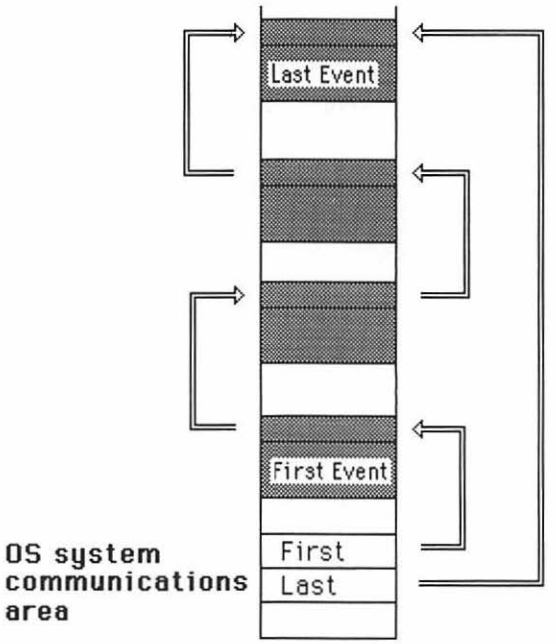
```
FlushEvents(eventMask, stopMask);
```

removes all events specified by "eventMask" from the event queue up to, but not including, the first of any type specified by "stopMask". Normally, we set eventMask to $FFFF and stopMask to $0000. This specifies that all events are to be removed until the event queue is empty. In the external file "ToolIntf", the value $FFFF is set equal to the constant "everyEvent", which is the first parameter for this function in our program.

If we use other values for these parameters, the Event Manager has to *selectively* remove items from the list. This is handled automatically by the Event Manager by manipulating the "qlink" pointers (see Figure 5-11).

The next procedure called in "Setup" is "SetEventMask". This is not a ROM routine. It is contained in a Pascal external file linked to your applications program. It merely places a specified eventMask in an Operating System variable called "SysEventMask". The Event Manager then uses this mask to determine the types of events to be placed on the event queue. In our program, the value "everyEvent" specifies that every type of event be placed on the event queue.

Figure 5-9. Event Queue Header



Making Titles

The next procedure in our program is called "MakeTitle". It draws a specified title at the top center of the active area of the current grafPort (as specified by its port rectangle). It has a single argument that is a dynamic string of type Str255, the standard type of string used by QuickDraw.

The "MakeTitle" procedure begins by a WITH statement, allowing us to specify the various fields ("top", "left", "bottom", and "right") of the current port rectangle. This shortens our formulas and makes them easier to read. In the WITH statement, we "MoveTo" a position that we compute for the beginning of the title.

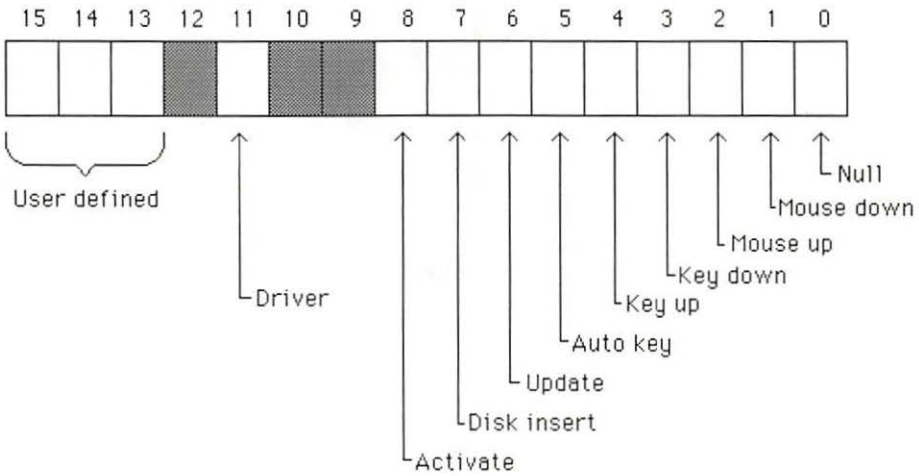
The "MoveTo" procedure is a QuickDraw routine that moves the graphics pen around the screen without drawing anything. However, in combination with the "LineTo" routine, it can help draw any line or combination of lines on the screen.

We use the following formula for computing its horizontal component:

```
(left + right - StringWidth(title)) DIV 2
```

This is equivalent to taking the midpoint of the screen and subtracting half of the physical length of the string on the screen. The QuickDraw function "StringWidth" computes this length as the number of pixels that are consumed in the horizontal direction by the string. The result will

Figure 5-10. The Event Mask



depend on the font, size, and style of the text. The "StringWidth" function is very powerful because it physically sizes up your text, taking into account all these considerations — including proportional spacing.

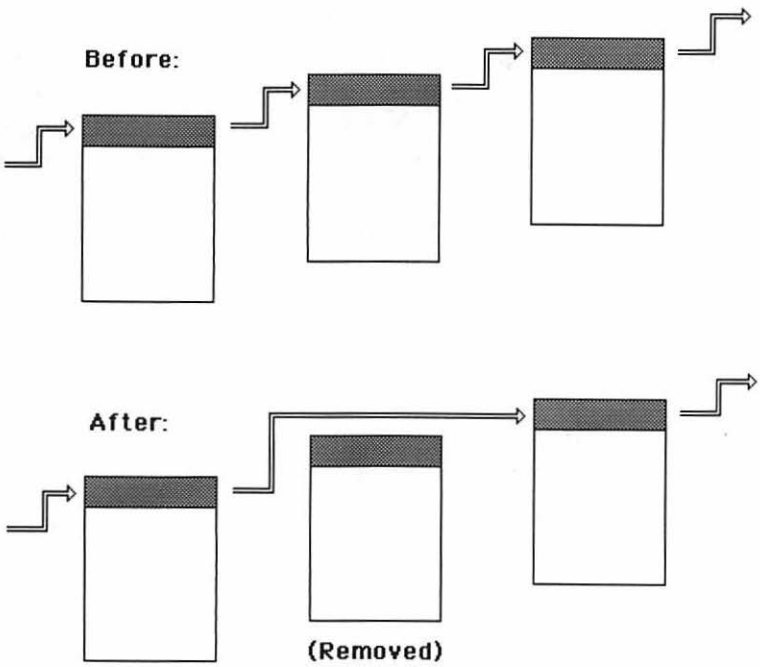
The vertical position of the text is twenty units from the top of the screen. This has to be adjusted for large text sizes. Note that the text is positioned so that its lower left corner falls at the current position (see Figure 5-12). Thus, the text stretches upward and to the right of the current position.

The QuickDraw "DrawString" routine draws the text on the screen.

Making Titled Rectangles

The next procedure is called "MakeRect". In our program, it makes rectangular boxes to display keyboard information. It has two parameters: a rectangle, "R", and a title, "title". It erases the rectangle, frames it, then draws a title above it.

Figure 5-11. Removing an Event from the Queue



Making Rounded Rectangular Regions

Next is the function "MakeRRgn". It makes three rounded "mouse boxes" that are implemented as *regions*. The "R" in the middle of its name stands for "Rounded". It has two parameters: a rectangle, "R", to determine the size of the box, and a variable, "Curv", of type "Point" to specify the roundedness or curvature of corners of the region (see Figure 5-13). The "MakeRRgn" procedure returns a region handle that allows you to properly access the newly created region.

The procedure calls "NewRgn" to allocate space for the region. This returns a value for a region handle that we temporarily store in the local variable "Rgn".

The "OpenRgn" routine allows us to "open" or start the region using QuickDraw routines to define the shape of the region.

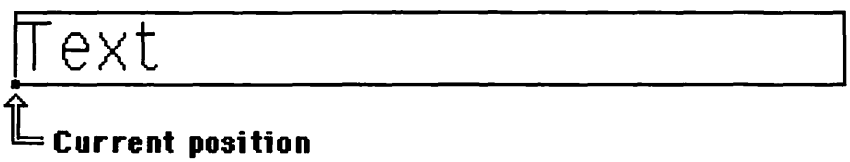
Instead of drawing to the screen when a region is "open", QuickDraw stores (records) "corner" data into a special "save" region whose handle is one of the fields of the grafPort. Chapter 4 describes how regions are stored in memory.

We issue only one QuickDraw command, "FrameRoundRect", to define this region. The parameters to the "FrameRoundRect" command are the rectangle "R" to determine the basic size of the rounded rectangle, the value "Curv.h" to determine the width of its corner oval, and the value "Curv.v" to determine the height of its corner oval (see Figure 5-13).

The region is "closed" with the "CloseRgn" statement. It is the opposite of "opening" the region. "CloseRgn" stops QuickDraw from drawing into the grafPort's "save" region and causes subsequent drawing to appear on the screen. This routine has a single parameter, a region handle that is a handle to the newly created region. In our program, the parameter is "Rgn", initialized at the beginning of this procedure.

Finally, we assign this handle value to the identifier "MakeRRgn" so that it is returned as the value of the "MakeRRgn" function.

Figure 5-12. Positioning Text



Drawing the Mouse Boxes

The final procedure in our program is "DrawRRgn". It draws the mouse boxes. It has two parameters: a region handle "Rgn" and a string, "title", of type Str255. The region handle specifies the region to be drawn, and the string specifies the title.

Within the routine, the "EraseRgn" routine erases the region, the "FrameRgn" routine frames it (draws a line around it), and the "DrawString" routine places the title above it. Before drawing the title, we use the "MoveTo" routine in a WITH statement to move the current position to where we want the lower left corner of the title.

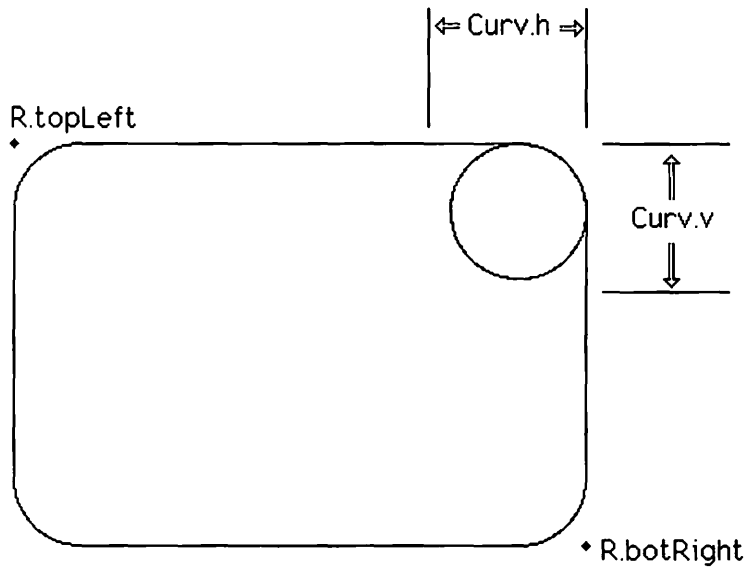
The Main Program

The main program consists of an initialization stage and a main REPEAT loop.

Initialization

The initialization stage begins by calling the "Setup" procedure to initialize QuickDraw, the Font Manager, and the Event Managers. It then calls

Figure 5-13. Specifying Roundedness



"MakeTitle" to place the string "Keyboard and Mouse Events" centered at the top of the screen.

Defining the Screen Layout

Next we define the rectangles that size all of our boxes, and the points that define the curvatures for our rounded rectangular regions. The QuickDraw routine "SetRect" defines the rectangles, and the "SetPt" routine defines the points. These statements are grouped in one place, making it easy for a programmer to control the layout of the screen. In Chapter 6, we explore how the Macintosh allows you to define screen layouts in separate *resource* files.

Setting up the Boxes

The next part of the program sets up the boxes on the screen. It calls the various procedures and functions defined earlier.

First, we make the box for the keyboard array by calling "MakeRect", passing it the keyboard rectangle "kbRect" and the title "Keyboard Array". We use the "VideoAddr" function to initialize the pointer "theKeys" so that it points to a spot in the box labeled "Keyboard Array" on the video screen where we want the display of the bits of the keyboard matrix to begin (see Figure 5-14). The "VideoAddr" function already knows where the screen is. Thus it can compute the proper location in video memory for any given point on the screen. Here, we specify a point relative to the upper left corner of the "kbRect" rectangle that defines the display box. Notice that the POINTER function converts the numerical value of the address to a pointer value. There is no conflict of types in the assignment of the pointer value returned from POINTER. In fact, pointer values returned from the POINTER function can be assigned to any pointer variable without complaint from the compiler.

Next, we set up the box for the key display by calling "MakeRect" with the parameters "keyRect" to define the size of the box and the literal string "Key" to title the box.

Now we set up the mouse points region. First, we use MakeRRgn to define a rounded rectangular region. This function returns a handle to the region assigned to the variable "MouseRgn". The size and position of this region is given by the first parameter, which is the rectangle "mouseRect"; its "curvature" is given by the second parameter, which is the point "mouseCurv". We then draw and label the region by calling our "DrawRRgn" procedure, passing the mouseRgn handle and the literal string "Mouse points" as the title.

The boxes to control mouse erasures and to exit the program are similarly set up, using the "MakeRRgn" function and "DrawRRgn" procedure.

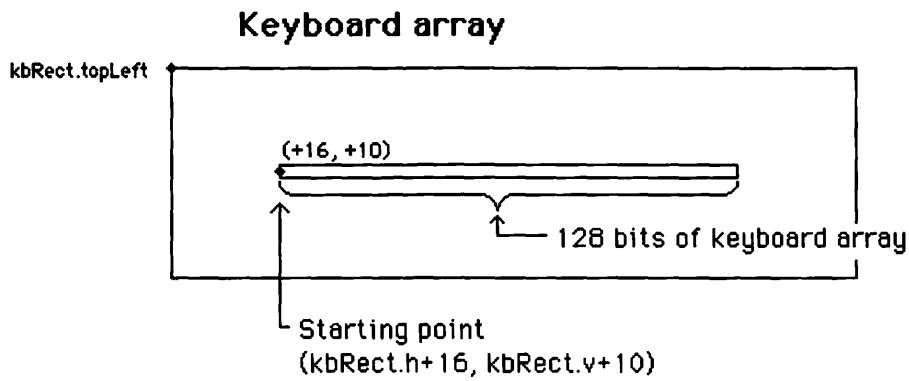
The Main Loop

The heart of the program is a REPEAT ... UNTIL loop called a *polling* loop. This loop continually gets events and handles them. Each time it gets the next event, it determines what occurred and performs an appropriate action. Many times there are no new events. In such a case, it will receive the "null" event, which it handles by doing no action except for getting the next event.

This *polling* method is in distinct contrast to the *interrupt* method that underlies the Macintosh's management of low-level I/O transactions. It is interesting to note that both methods coexist in the machine at once, interrupts at the low levels and polling at the high levels. An alternate approach would be to have interrupts at both levels. That would make programming more interesting and efficient but more difficult to understand and debug.

The main loop begins with a call to the Event Manager's "GetNextEvent" function. This retrieves the event record of the next event of specified types from the event queue. Its first parameter is an integer that holds an event mask specifying those events to be taken from the queue. The second parameter, the event record, is passed by reference (as a VAR parameter), since it is returned. The function value returned by "GetNextEvent" is a Boolean variable that indicates whether or not the appli-

Figure 5-14. Positioning the Keyboard Array



cations program should try to handle the event or leave it for the Operating System. That's why we encase "GetNextEvent" within an IF ... THEN statement. The THEN clause is executed if the event should be handled by the applications program.

The Cases

Once we get the event and decide to handle it, we use a CASE statement to determine what type of event occurred. The cases that we check are mouseDown, keyDown, and keyUp. Typically, applications programs handle more cases, but this program merely serves to introduce events, so it is as simple as possible.

Within the "mouseDown" case, we see if the mouse is in any of the three mouse boxes. First, we get the local coordinates of the mouse into the point variable MPt by using the Event Manager's "GetMouse" routine. The global coordinates of the mouse are available in the field "theEvt.where". However, we want the local coordinates. We could use the "GlobaltoLocal" routine to convert this field to local coordinates. This is done in other example programs, but not here. Recall that the global coordinates are attached to the screen, whereas local coordinates are attached to specific grafPorts. Later, when we study *Windows*, we will see how local coordinates define positions of objects (such as *controls*) relative to the windows that contain them.

We use the "PtInRgn" function to see if the mouse point is within each region. If the mouse is within the mouse points region, we plot a point there by applying the "MoveTo" routine to MPt, then invoke the "Line(0,0)" statement. This last statement is a relative line-drawing command that draws a line from the current position to itself, making a single point at the current position.

On the next line of the program, we see if the mouse is within the erase region. If it is, we redraw the mouse points box, which has the effect of erasing it.

Next we see if the mouse is within the exit region. If it is, we set the "done" variable to true. This terminates the REPEAT UNTIL done loop and ends the program.

The keyDown case displays the key within the key box. Here we first "MoveTo" a point within the key box. Then we plot the key character. The ASCII code for the key character is contained within the lowest eight bits of the message field of the event record. We use "MOD 256" to extract the ASCII code from the message field. Then we use the CHR function to convert it to a character and use the QuickDraw "DrawChar" routine to draw it.

The keyUp case clears the key box by redrawing it with our "MakeRect" procedure.

Displaying the Keyboard Matrix

At the bottom of the loop we display the keyboard matrix by invoking the Event Manager's "GetKeys" routine.

Generally, "GetKeys" has a single parameter of type "keyMap", which is passed by reference. The routine loads a copy of the keyboard matrix into this variable.

In this particular program, we pass to it the expression "theKeys^ ". Since we loaded the address of a point on the screen into "theKeys", this expression specifies that particular location in video memory. Since "theKeys" is of type "KeyPtr", the expression has the correct type.

Summary

This chapter has discussed how to write applications programs that use the Macintosh's Event Manager. We have studied the data structures and routines that interact with this Manager and make the Event Manager work.

We have seen the power of such programming. We have seen how it allows us to pick up keys and key combinations from the keyboard and how it allows us to track the mouse and easily determine when it is pressed in any area of the screen.

We have also touched on the concepts of controls and resources, which are explored in later chapters.

This chapter covers the following ROM rountines:

EM-FlushEvents

EM-SetEventMask

QD-OpenRgn

QD-FrameRoundRect

QD-CloseRgn

QD-EraseRgn

QD-FrameRgn

EM-GetNextEvent

QD-PtInRgn

QD-DrawChar

EM-GetKeys

6

Introduction to Windows

This chapter covers the following new concepts:

- **Windows**
- **Controls: Scroll Bars**
- **Window Manager**
- **Window Parts**
- **Window Updating**
- **Window Activation and Deactivation**
- **Tracking, Dragging, and Sizing Windows**
- **Standard Window Regions**
- **QuickDraw Text Attributes**

Windows add another dimension to an applications program. They allow users to handle a multitude of separate pieces of information according to the specifications and control of the user rather than just the programmer (see Figure 6-1).

Windows are an extension of the QuickDraw grafPort. Both are picture-drawing environments. However, windows have the friendly advantage of being easily and naturally moved on the screen and resized by the user.

In this chapter, we introduce the fundamentals of managing a window. We describe how to draw pictures in a window and explore the structure

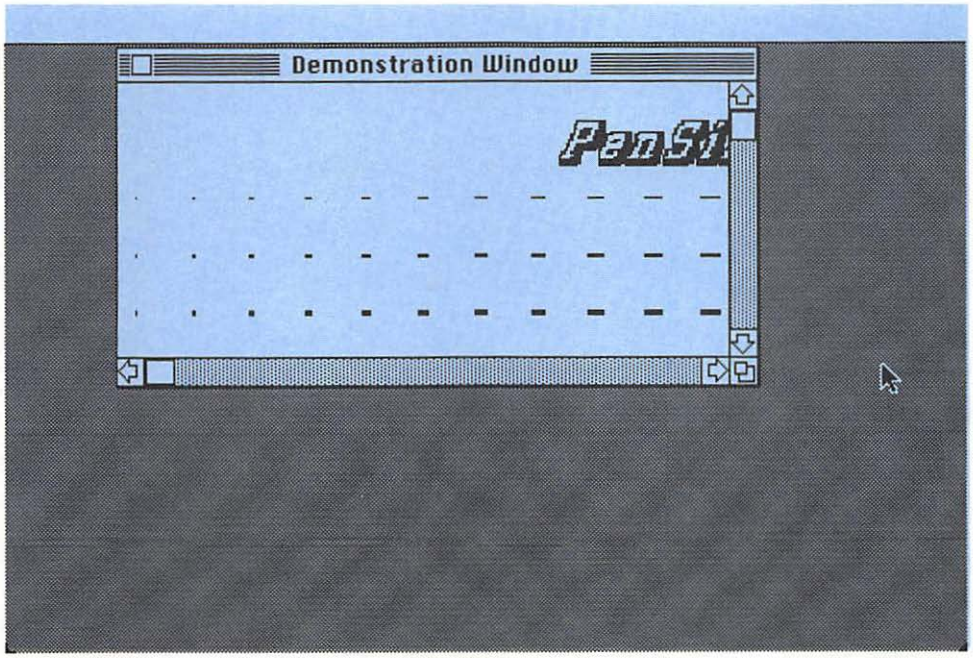
of a window. We present an example program to show how some window parts can control the size and position of the window on the screen.

We also introduce *controls*. These structures, attached both logically and physically to windows, allow the user to select values. Each control houses and manages a single *control value*. In our example program, we introduce a special kind of control called a scroll bar that houses scrolling values. We show how to scroll the contents of a window with these controls.

The basic parameters that define windows and controls are stored as *resources* along with the machine code that forms your program. In this chapter, we see how to set up and use such resources.

Window management is essentially performed by the collection of routines and data structures that form the *Window Manager*. However, this management requires the cooperation of the Event Manager, the applications program, and the Control Manager, as well as the Window Manager. The example program demonstrates the programming interrelationships of these different parts of the system.

Figure 6-1. A Window



Parts of a Window

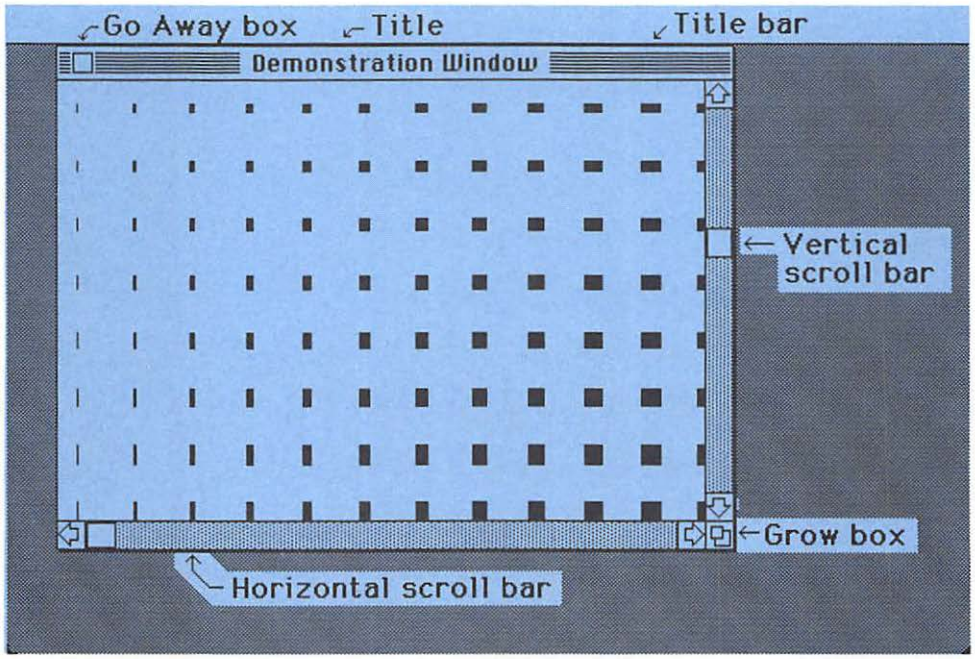
A window has several parts that serve specific functions (see Figure 6-2). These include the *frame*, the *title*, the *title bar*, the *goAway box*, the *grow box*, the *contents*, and the vertical and horizontal *scroll bars*. You are familiar with these parts as a Macintosh user.

The window *frame* consists of the title area containing the *title bar*, the *goAway box*, and the boundary lines around the window. The contents of a window include the area where the picture is drawn as well as the areas where the scroll bars and grow box appear.

Each scroll bar consists of several parts, including the *up button*, the *down button*, the *page up* control, the *page down* control, and the *thumb* control (see Figure 6-3).

The window frame, including the title, title bars, and goAway box, is drawn automatically by the Window Manager whenever the window is moved or resized. The grow box is redrawn by the Window Manager on special request. The scroll bars are drawn by the Control Manager on request by the applications program. We see how these requests are made as we continue.

Figure 6-2. Parts of a Window



The Example Program

The example program displays a single window on the screen entitled "Demonstration Window". When the program is executed, you see in this window the upper left portion of an illustration of different pen sizes. The title "PenSizes" with large, outlined italic lettering is partly visible.

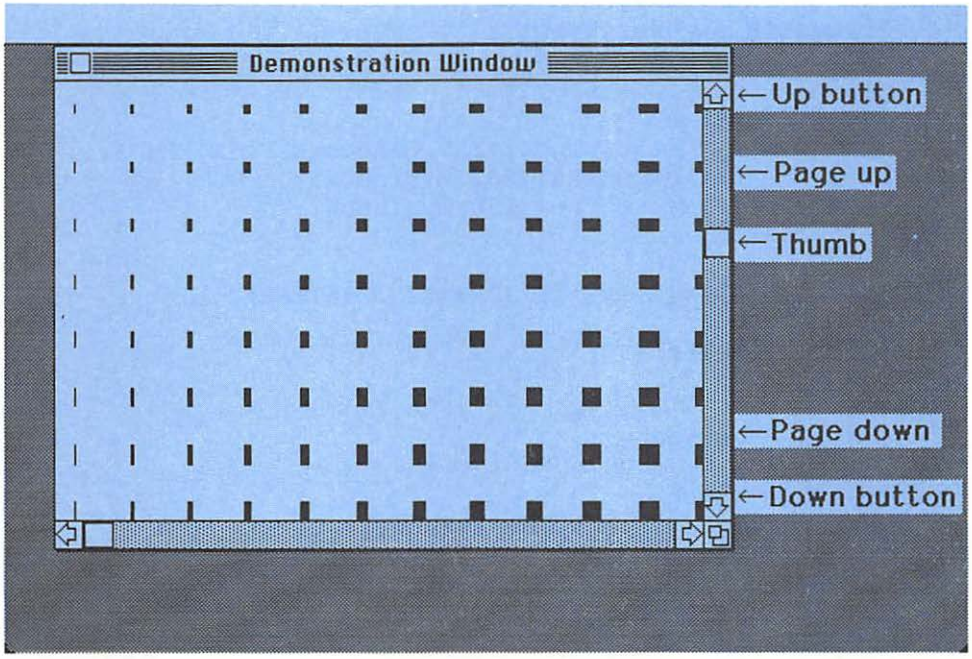
The entire illustration is too big to fit the screen. We have to use the scroll and size controls to see each section of this picture.

All basic parts of a typical window are present and working. Assuming that you have typed in and compiled the program, let's try exercising them.

First, try dragging the window to a new position, using the title bar to grab hold of it. Now try changing its size by dragging the size box. It grows to its new size when you release the mouse button.

Next, try the scroll controls. Use the thumbs to scroll to any horizontal or vertical position. Use the page up and page down controls to move by entire pages, and use the up and down buttons to move one row or column at a time. Notice that the horizontal scroll bar directions are "left" and "right".

Figure 6-3. Parts of a Scroll Bar



Finally, try selecting the "goAway" button to end the demonstration. Now let's examine this program.

```
PROGRAM WM;
   {$R-}{$X-}
   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf;

   CONST
      isize = 30;

   VAR
      done : BOOLEAN;
      theEvt : eventRecord;
      wRecord : windowRecord;
      myWindow, theWindow : windowPtr;
      dragBnds, sizeBnds, drawRect, clipBnds: Rect;
      hsbar, vsbar : controlHandle;

   PROCEDURE Setup;
      BEGIN
         InitGraf(@thePort);
         InitFonts;
         FlushEvents(everyEvent,0);
         SetEventMask(everyEvent);
         InitCursor;

         InitWindows;
         myWindow := GetNewWindow(257,@wRecord,POINTER(-1));
         hsbar := GetNewControl(257,myWindow);
         vsbar := GetNewControl(258,myWindow);
      END;

   PROCEDURE PenShapes(x, y, MaxI, MaxJ : INTEGER);
      VAR
         I, J : Integer;

      BEGIN
         EraseRect(drawRect);

         TextFont(Athens);
         TextFace([italic,outline,shadow]);
         TextSize(24);
         MoveTo((isize*MaxI-StringWidth('PenSizes')) DIV 2-x, 40-y);
         DrawString('PenSizes');
```

```
      MoveTo(10 - x, 60 - y);
      FOR J := 1 to MaxJ DO
         BEGIN
            FOR I := 1 to MaxI DO
               BEGIN
                  PenSize(I, J);
                  Line(0, 0);
                  Move(isize, 0);
               END;
            Move(-isize*MaxI, isize);
         END;
      PenNormal;
   END;

PROCEDURE ShowPict(theWindow: WindowPtr);
   BEGIN
      WITH theWindow^.portRect DO
         SetRect(clipBnds, left, top, right-15, bottom-15);
      ClipRect(clipBnds);
      IF theWindow = myWindow THEN
         PenShapes(GetCtlValue(hsbar),GetCtlValue(vsbar),20,20);
      ClipRect(drawRect);
   END;

PROCEDURE WindowGrow(theWindow: WindowPtr; thePt: Point);
   VAR
      WSize : LONGINT;
      S : Point;
   BEGIN
      WSize := GrowWindow(theWindow,thePt,sizeBnds);
      IF WSize = 0 THEN Exit(WindowGrow);

      SetPt(S,loWord(WSize),hiWord(WSize));
      SizeWindow(theWindow, S.h, S.v, true);
      DrawGrowIcon(theWindow);
      SizeControl(hsbar, S.h-13,     16);
      MoveControl(hsbar,     -1, S.v-15);
      SizeControl(vsbar,     16, S.v-13);
      MoveControl(vsbar, S.h-15,     -1);
   END;

PROCEDURE ScrAction(theCtl: ControlHandle; partCode: INTEGER);
   VAR
      pagesize, delta : INTEGER;
   BEGIN
      WITH thePort^.portRect DO
         CASE GetCRefCon(theCtl) OF
            1: pagesize := right - left - 16;
            2: pagesize := bottom - top - 16;
```

```
            otherwise    Exit(ScrAction);
            END;
         CASE partCode OF
            inUpButton:    delta := -isize;
            inDownButton:  delta := +isize;
            inPageUp:      delta := -pagesize;
            inPageDown:    delta := +pagesize;
            otherwise      Exit(ScrAction);
            END;
         SetCtlValue(theCtl,GetCtlValue(theCtl)+delta);
         ShowPict(thePort);
      END;

   PROCEDURE WindowScroll(theWindow: WindowPtr; thePt: Point);
      VAR
         theCtl : ControlHandle;
      BEGIN
         SetPort(theWindow);
         GlobalToLocal(thePt);
         CASE FindControl(thePt,theWindow,theCtl) OF
            inUpButton, inDownButton, inPageUp, inPageDown:
               IF TrackControl(theCtl,thePt,@ScrAction)<>0 THEN;
            inThumb:
               IF TrackControl(theCtl,thePt,NIL) <> 0 THEN
                  ShowPict(theWindow);
            END;
      END;

   PROCEDURE WindowUpdate(theWindow: WindowPtr);
      BEGIN
         SetPort(theWindow);
         BeginUpdate(theWindow);
            InvertRect(theWindow^.portRect);
            SysBeep(10);
            DrawControls(theWindow);
            DrawGrowIcon(theWindow);
            ShowPict(theWindow);
         EndUpdate(theWindow);
      END;

   BEGIN {main program}
      Setup;
      SetRect(drawRect,   0,   0, 512, 342);
      SetRect(sizeBnds,  50,  50, 512, 342);
      SetRect(dragBnds,   4,  24, 508, 338);
```

```
    done := false;
    REPEAT
       IF GetNextEvent(everyEvent, theEvt) THEN
          CASE theEvt.what OF
             mouseDown:
                CASE FindWindow(theEvt.where, theWindow) OF
                   inContent:
                      WindowScroll(theWindow, theEvt.where);
                   inDrag:
                      DragWindow(theWindow, theEvt.where, dragBnds);
                   inGrow:
                      WindowGrow(theWindow, theEvt.where);
                   inGoAway:
                      done := TrackGoAway(theWindow, theEvt.where);
                   END;
             updateEvt, activateEvt:
                WindowUpdate(POINTER(theEvt.message));
          END; {what event}
    UNTIL done;
END.
```

External Files

The USES section of this program requests the same external files as the example program in Chapter 5. The new data structures for windows and controls are located in the file "ToolIntf". However, we are building and drawing in our window using QuickDraw and the Event Manager, so we need the other external files as well.

Constants

The CONST section of this program defines only one contant: "isize". This defines the spacing within the diagram and helps control scrolling. It globally affects our program and thus belongs in the global constants section.

Global Variables

The VAR section contains the Boolean variable "done" and the Event Record "theEvt", which manage events in much the same manner as described in Chapter 5.

Window Records

In addition to event management variables, our program has a number of global variables to program our window. The first variable is "wRecord", which is a *Window Record* (Pascal data type "windowRecord").

In this section, we study this structure in detail. It contains all the information needed by the Window Manager to manage one window. A *window* really consists of three entities: the *image* of the window on the screen, the *Window Record* (studied in this section), and the Window Manager routines (introduced in this chapter).

Other Macintosh concepts, including *controls*, *dialogs*, *alerts*, and *menus*, behave in the same three-part way. For each concept, we describe an image on the screen, a data structure, and a set of routines. As with windows, complete understanding and control of these concepts requires a detailed knowledge of each part, including the data structures in memory.

Let's continue our discussion of windows. The data stored in a Window Record include a grafPort and a number of Boolean variables, pointers, and handles required to manage the window. These data are central to understanding how the Macintosh manages multiple overlapping windows.

Let's look closer at this data structure:

```
WindowRecord =
   RECORD
      port:           GrafPort;
      windowKind:     INTEGER;
      visible:        BOOLEAN;
      hilited:        BOOLEAN;
      goAwayFlag:     BOOLEAN;
      spareFlag:      BOOLEAN;
      strucRgn:       RgnHandle;
      contRgn:        RgnHandle;
      updateRgn:      RgnHandle;
      windowDefProc:  Handle;
      dataHandle:     Handle;
      titleHandle:    StringHandle;
      titleWidth:     INTEGER;
      controlList:    Handle;
      nextWindow:     WindowPeek;
      windowPic:      PicHandle;
      refCon:         LongInt;
   END;
```

The first field, ".port", is the window's grafPort. As discussed in Chapter 4, this contains drawing variables to specify the size and shape

of the drawing area; the pen size, pattern, and mode; and the text size, font, and face.

The second field, ".windowKind", is an INTEGER that classifies the window. Currently, there are two supported kinds: a value of two indicates that the window is used as a dialog or alert, and a value of eight indicates that the window is a normal user-created window.

The third field, ".visible", is a BOOLEAN that specifies if the window is visible.

The fourth field, ".hilited", is a BOOLEAN that specifies if the window is highlighted.

The fifth field, ".goAwayFlag", is a BOOLEAN that specifies if the window has a "goAway" box.

The sixth field, ".spareFlag", is a BOOLEAN reserved by Apple for future use. It is included because of memory alignment considerations. More precisely, the previous three fields are Boolean variables, each taking one byte of memory, placing us in the next field at an odd address. The "spare" flag adds another byte so that the following field has an even-numbered address, as required by the 68000 processor to access integers and long integers.

The seventh field, ".strucRgn", is a region handle (stored as a long integer) to the *structure region* of the window. This region delimits the entire window.

The eighth field, ".contRgn", is a region handle to the content region of the window. This is the area where drawing, the grow icon, and the scrolling controls are placed (see Figures 6-1, 6-2, and 6-3).

The ninth field, ".updateRgn", is a region handle to the update region. This region accumulates areas of the window for updating. We study updating later.

The tenth field, ".windowDefProc", is a handle to the window definition procedure. This procedure performs a variety of functions, such as drawing the window frame, returning the region that the mouse was pressed in, calculating the structure and content regions, drawing the grow icon, and performing any needed initialization and disposal. When the Window Manager calls this routine, it sends a code which specifies the required action. Programmers can write and install window definition procedures to build custom windows. (This is beyond the scope of this book, but Apple manuals show how.) Programmers should use this capability wisely, for users expect applications to behave in a consistent manner. Apple has provided guidelines, but programmers should play with the system and other applications programs to get a feel for what users want and expect.

The eleventh field, ".dataHandle", is a handle to data that the window definition procedure may need.

The twelfth field, ".titleHandle", is a StringHandle that leads to the title of the window. StringHandles are defined by the following data structures:

```
StringPtr    = ^Str255;
StringHandle = ^StringPtr;
```

That is, they are handles to the standard type of strings used by QuickDraw.

The thirteenth field, ".titleWidth", is an INTEGER that gives the width (in pixels) of the window's title.

The fourteenth field, ".controlList", is a handle to the list of controls that belong to a window. This is a linked list that uses handles instead of pointers. In our program, our window points to its scroll bars via this field (see Figure 6-4).

The fifteenth field, ".nextWindow", is of type "WindowPeek" and points to the "next" window. The Window Manager maintains a linked list of all windows. A variable, "WindowList", in the Toolbox's data area points to the first window, and the linking is implemented by the ".nextWindow" field of each window.

The "WindowPeek" data type is defined here by the Pascal type declaration:

```
WindowPeek = ^WindowRecord;
```

That is, it points to the entire window record. In contrast, a type "WindowPtr" is defined by the Pascal type declaration:

```
WindowPtr = GrafPtr;
```

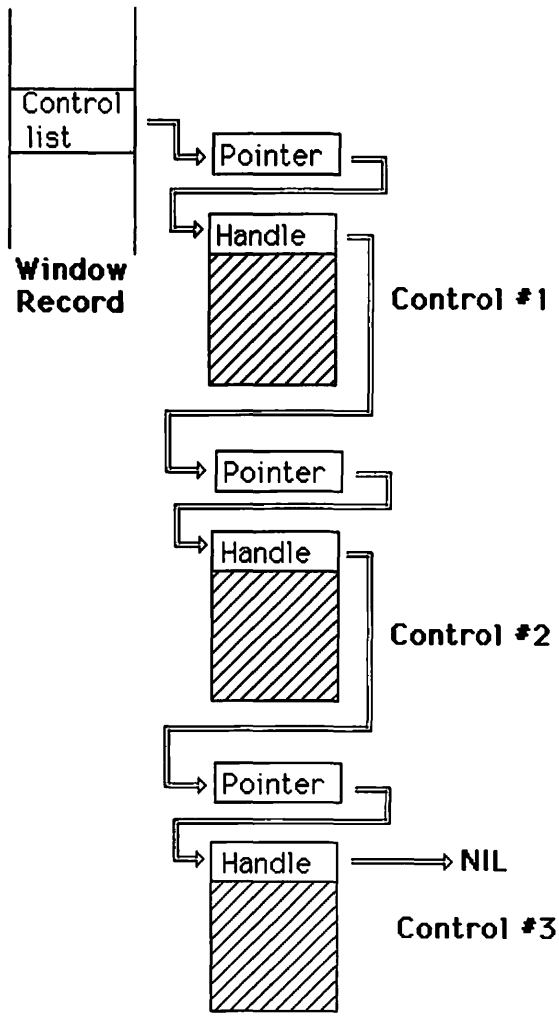
which accesses only the grafPort of the window, denying access to the window's remaining data structure. The more limited "WindowPtr" is the type normally used by an applications programmer, whereas "WindowPeek" is typically used by the system. The Window Manager provides access to most fields of the window record by its routines. Thus, it is not necessary to access these fields directly. Using indirect methods such as routines to access data is a modern approach that provides better protection of vital data than more direct methods.

The sixteenth field, ".windowPic", is a PicHandle. In Chapter 7, we introduce the pictures concept and explain this field.

The seventeenth field, ".refCon", is a LongInt. This field can be used for any purpose. The programmer can use the "SetWRefCon" procedure to place numerical values in this field and the "GetWRefCon" to retrieve them. For example, programmers can use this field to hold the numerical value of a handle to another structure such as a control or a region, the address of a routine, or perhaps an index to an array.

In this program, we use the static variable "wRecord" to store the data for our single window. It is possible to store this structure dynami-

Figure 6-4. The Control List



cally. It must then be stored in a nonrelocatable block of memory, since it is accessed by a pointer. Only data accessed by handles can be stored in relocatable blocks of memory (see Chapters 3 and 4). This can cause problems in large applications programs because nonrelocatable areas on the heap tend to "fragment" the heap, making it hard for the Memory Manager to remove "holes" by "compacting" memory.

Some Window Pointers

The next global variables are "myWindow" and "theWindow". These are of type "windowPtr" and point to our window. The first defines the window, and the second is used in conjunction with the "FindWindow" routine, which identifies the window that the mouse is in. Since there is only one window, you might think this is always "myWindow". But sometimes the mouse cursor is in no window. In this case, "FindWindow" returns a NIL pointer. We definitely need another variable to accept this value when it occurs; otherwise we might lose track of our window.

Window Limits

The next four global variables are rectangles to help control dragging, sizing, drawing, and clipping of our window.

The first rectangle, "dragBnds", describes how far (in global coordinates) the top left corner of the window's content region can be dragged (see Figure 6-5). We don't want the window to be dragged completely off the screen. Later in the program, we specify appropriate values for the corner points of this rectangle.

The second rectangle, "sizeBnds", sets limits on the size of the content region when we "grow" the window (see Figure 6-6). The ".topleft" corner point is the minimum value and the ".botright" corner point is the maximum value for its vertical and horizontal size. If a window is too small, we may have difficulty finding its controls and may lose control of it.

The third rectangle, "drawRect", describes the drawing area. This may be much larger than the window and is related more to the picture than the window. In Chapter 7, it figures importantly in conjunction with pictures.

The fourth rectangle, "clipBnds", specifies the area of the window's content region where the picture is seen. This is smaller than the entire content region because it excludes those areas of the content region where scroll bars and the grow icon appear. We have to recompute this rectangle at least every time the picture is resized.

Controls

The last two global variables are the *control handles* to manage the scrolling controls. The "hsbar" is a handle to the horizontal scroll bar, and the "vsbar" is a handle to the vertical scroll bar.

Figure 6-5. Dragging Limits

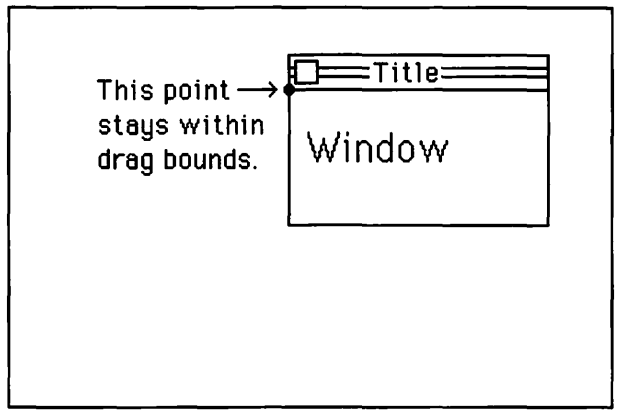
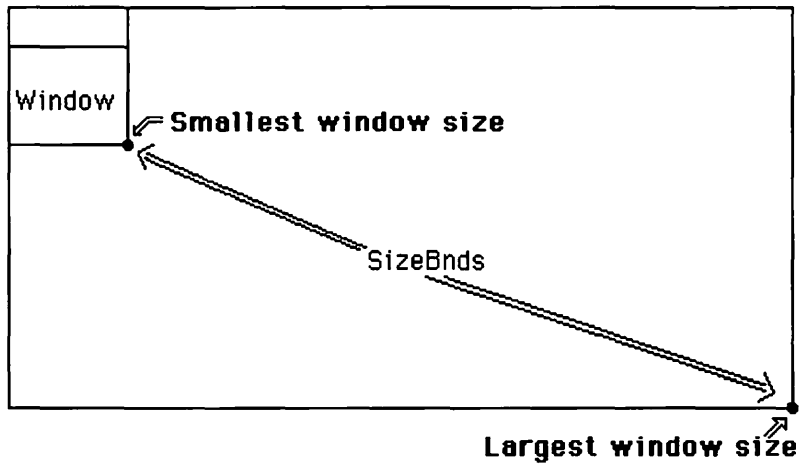


Figure 6-6. Sizing Limits



These are the first of many *controls* studied in this book. In Chapter 7, we introduce *standard button* controls, and in Chapter 8, we introduce *check boxes* and *radio buttons*.

A control allows the user to easily manage a single value. In our program, a *horizontal displacement* is stored in the data structure for the horizontal scroll bar, and a *vertical displacement* is stored in the data structure for the vertical scroll bar. These displacements are fed into the picture-making procedures to offset the position of the picture from the window. Scrolling is accomplished by drawing the picture as these displacements change.

Like windows, *controls* consist of three parts: the image of the control on the screen, the control's data structures, and the Control Manager routines that operate on the control. As with other Macintosh concepts, studying the control's data structure is a key to understanding how it works and what it is capable of.

A control's data structures are defined by the following series of Pascal type declarations:

```
ControlHandle = ^ControlPtr;
ControlPtr    = ^ControlRecord;

ControlRecord =
   RECORD
      nextControl:   ControlHandle;
      contrlOwner:   WindowPtr;
      contrlRect:    Rect;
      contrlVis:     BOOLEAN;
      contrlHilite:  BOOLEAN;
      contrlValue:   INTEGER;
      contrlMin:     INTEGER;
      contrlMax:     INTEGER;
      contrlDefProc: Handle;
      contrlData:    Handle;
      contrlAction:  ProcPtr;
      contrlRCon:    LongInt;
      contrlTitle:   Str255;
   END;
```

That is, a control handle points to a control pointer, which points to a control record.

Thus this structure contains fields to help manage the control's value, define how it is drawn, and connect it to other structures.

As discussed, each window has a handle to a list of controls (see Figure 6-4 above). This is a linked list in which linking is by handles

rather than pointers. The first field, ".nextControl", of each control record is a control handle. This control handle provides the linking to the next control in the list. The last control in a window's control list always has a NIL (zero) value in this field.

The second field, ".contrlOwner", of a control record is a window pointer that points to the window record of the window that "owns" the control. Reciprocally, the "owner" window has the control in its control list (see Figure 6-7).

The third field, ".contrlRect", is a rectangle that delimits the area that the control occupies within the window (see Figure 6-8). The control may occupy an irregular region smaller than this rectangle.

The fourth field, ".contrlVis", is a BOOLEAN that specifies if the control is visible.

The fifth field, ".contrlHilite", is a BOOLEAN that specifies if the control is highlighted. In Chapter 7, we exercise this feature.

The sixth field, ".contrlValue", is an INTEGER containing the current value of the control.

The seventh field, ".contrlMin", is an INTEGER that specifies the minimum value of the control.

The eighth field, ".contrlMax", is an INTEGER that specifies the maximum value of the control.

The ninth field, ".contrlDefProc", is a handle to the control's definition procedure. This procedure performs functions such as drawing the control, testing for the mouse point in the control, calculating the control's region, and updating the control's appearance. Programmers can install their own routines here. The earlier comments about custom window definition procedures apply here as well (see earlier ".windowDefProc" discussion).

The tenth field, ".contrlData", is a handle to data for a control's definition procedure.

The eleventh field, ".contrlAction", is a pointer to the control's action procedure. This procedure tracks controls.

The twelfth field, ".contrlRCon", is a LongInt that can be used for any purpose, like the " .refCon" field of a window.

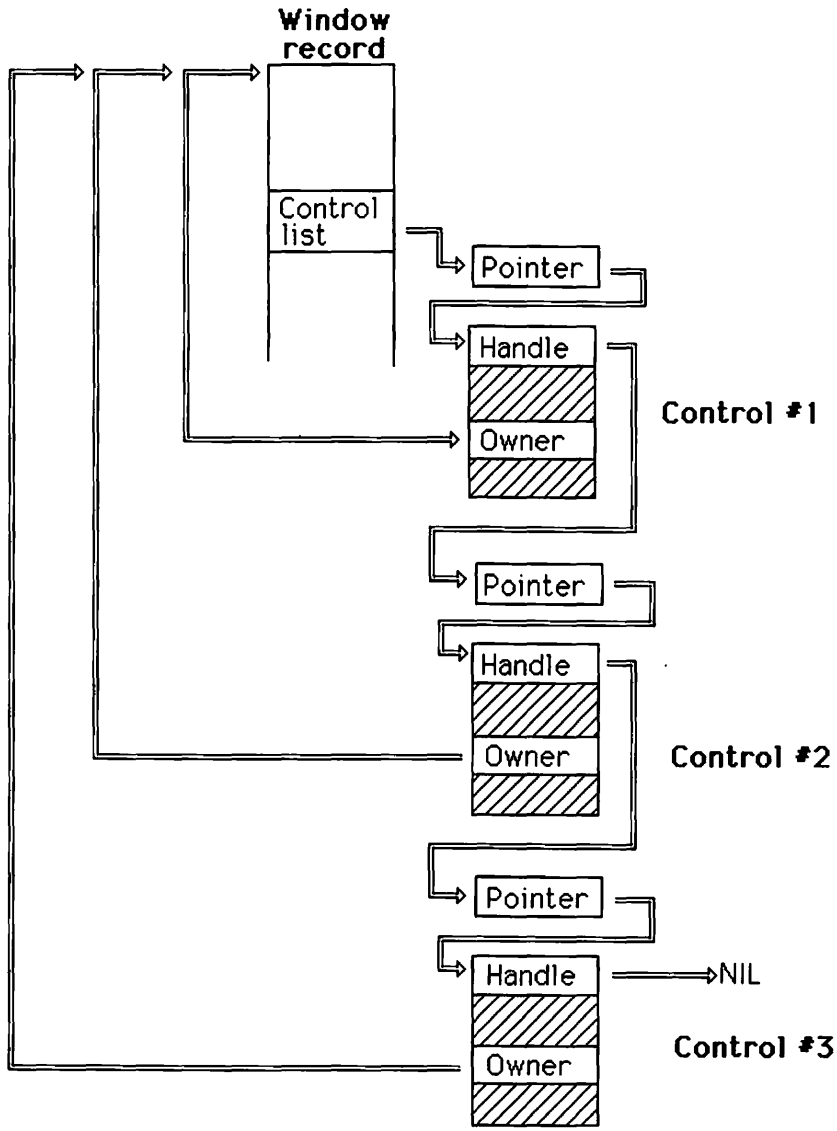
The thirteenth field, ".contrlTitle", is a dynamic string of type "Str255" that contains the title.

Procedures and Functions

Now that we've explored the data structures in our program, let's examine its procedures.

This program includes procedures and functions to initialize, to draw the figure that appears in the window, to "mount" the figure in the window, to resize the window, and to scroll the contents of the window.

Figure 6-7. Ownership of Controls



The Setup

The first procedure is "SetUp". It initializes the various managers, including QuickDraw, the Font Manager, the Event Manager, and the Window Manager.

The first part of this routine is similar to the "SetUp" routine in Chapter 5. However, here we do not open a grafPort: this is done automatically when we initialize the Window Manager.

Initializing the Window Manager

The second part of the routine initializes the Window Manager, then sets up our window with its two scrolling controls.

The first command, "InitWindows", initializes the Window Manager, setting up a grafPort to cover the entire screen. This grafPort belongs to the Window Manager itself, not to any of its windows.

Setting up the Window

The next command:

```
myWindow := GetNewWindow(257,@wRecord,POINTER(-1));
```

sets up a window defined as a *resource* in the application's disk file.

The "GetNewWindow" routine draws the window frame and tells the Event Manager to generate a special *event* called an *update event*. An *update event* tells the applications program to draw the rest of the window as part of its regular window maintenance cycle. Later, we see how the applications program detects and responds to window update events.

Figure 6-8. Control Rectangle

The "GetNewWindow" routine has three parameters: an integer "windowID" that is the resource identification number for the window, a pointer "wStorage" that points to the area of memory where the window's window record is stored, and a window pointer "behind" that helps locate the window's "depth" in relation to other windows.

Window Resources

Now let's examine window resources. Chapter 3 discussed how each applications program consists of a collection of resources. The program's code is one such resource; other resources include constants such as initial values for the fields of windows and controls. For our example programs, a *resource definition file* is like source code for these resources. It names the file where the raw machine code is located and explicitly gives window, control, and other resource definitions. When the application is processed into a file that is executable on the Macintosh, these resources are brought together.

The window identification number specifies a resource of type "WIND" in our resource file that should contain the defining parameters for our particular window. We use a value of 257 for our identification number. It is larger than 256 to avoid any hint of conflict with the system's identification numbers.

The following section of our resource definition file governs windows:

```
Type WIND
  ,257
  Demonstration Window
  40 60 200 400
  Visible GoAway
  0
  0
```

The first line is a *type* statement. It declares that the next item or items in the file (until we get to the next type statement) are definitions of windows.

The next line is the first line of our window resource definition. It contains a comma followed by the resource's identification number.

The second line of the resource definition specifies its title. Here, we use the title "Demonstration Window".

The third line of the resource gives the global coordinates of the corners of a window's port rectangle.

The fourth line specifies the visibility of the window and the presence of the "goAway" box. Here, we specify the window to be "visible" (instead

of "hidden") and to have a "goAway" box (instead of "noGoAway" for no such box).

The fifth line gives the window definition procedure's identification number. Here, we select zero, indicating to the Window Manager that it should use its procedure for a standard document window. Table 6-1 shows the choices for this parameter.

The final line gives the initial value of the window's reference value. Remember that this number is available for the programmer's use. We place a zero here because we are not using this particular field.

In "wStorage", the second parameter of the "GetNewWindow" routine, we place "@wRecord". This points to the static window record, a global variable. If the system is to allocate storage for the window record, we place a NIL value in this slot. Be aware that such storage is "nonrelocatable": it is accessed only by a pointer, not a handle. This may make a difference if you are short on memory.

The third parameter, "behind", points to the window that the new window is placed behind in the system's list of windows. This indicates how the windows appear on the screen. In this case, "POINTER(-1)" places it behind no other windows; that is, in front of all others. A value of NIL (that is, POINTER(0)) places it behind all other windows. In the case of this program, the particular value doesn't matter because there is only one window.

The "GetNewWindow" function allocates (if necessary) and initializes the window record, then returns a window pointer that points to this window record. In this case, we assign the value of this pointer to the variable "myWindow" so that "myWindow" now points to the newly created window.

Table 6-1. Window Definition Procedure Numbers

Identification Number	*Window Type*
0	Standard document window
1	Alert or dialog box
2	Plain box
3	Plain box
4	Document window without size box
16	Rounded-corner window

Setting up Controls

Once we have fetched the window, we set up its scrolling bar controls with the following statements to invoke the "GetNewControl" function:

```
hsbar := GetNewControl(257,myWindow);
vsbar := GetNewControl(258,myWindow);
```

The "GetNewControl" function has two parameters: an integer to specify a resource identification number for the control and a window pointer to specify the window that "owns" the control. The function returns a handle to the newly created control.

Here, we define the control with resource identification numbers 257 and 258. Again, we pick numbers larger than 256 to avoid conflict with the system's identification numbers.

Control Resources

The section of the resource file for these controls appears as follows:

```
Type CNTL
  ,257
  horizontal scroll bar
  145 -1 161 326
  Visible
  16
  1
  0 0 500

  ,258
  vertical scroll bar
  -1 325 146 341
  Visible
  16
  2
  0 0 500
```

The first line declares that the following items (until the next "type" statement) are resources of type "CNTL", the resource type for controls. The first line of each control resource contains its resource identification number preceded by a comma. Here, we have resources 257 and 258.

The second line of each control resource contains the control's title. In our case, we use the title merely as documentation to label the resource in the resource file, since scroll bars do not actually display titles.

The third line specifies the control's bounds rectangle, expressed in local coordinates of the window to which it belongs. In our case, we carefully choose numbers to place the scroll bars where they belong relative to the initial size of our window (see Figure 6-9). (The method for getting the correct placement is called "trial and error".)

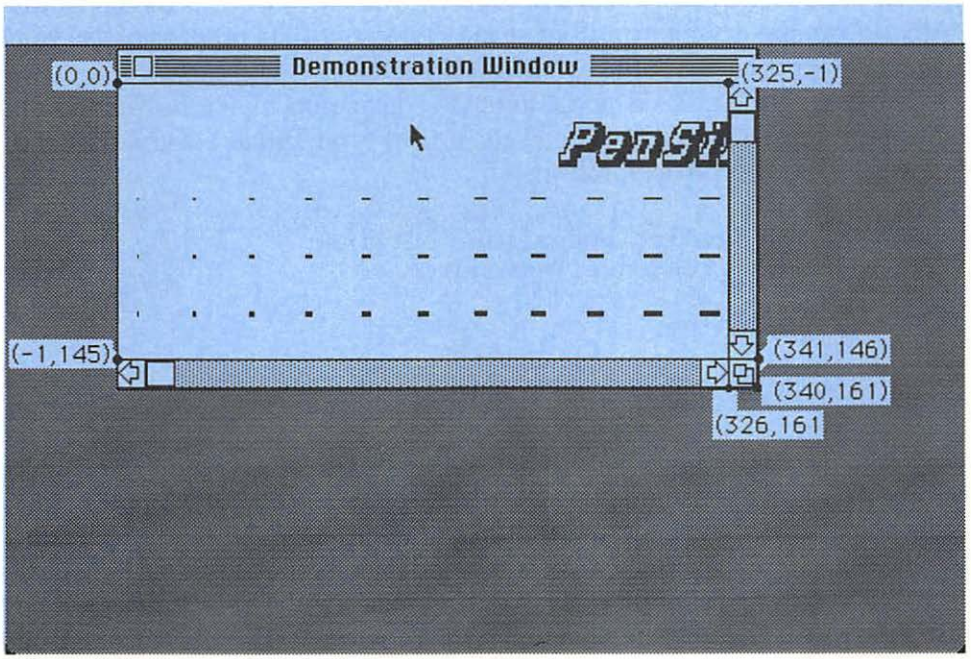
The fourth line specifies whether the control is "visible" or "hidden". Both our scroll bars are "visible".

The fifth line specifies the control's procedure definition identification number (see Table 6-1). We select 16 to identify the procedure that draws scroll bars.

The sixth line specifies the initial value of the control's reference value. Here, the reference value for the horizontal scroll bar equals one and the reference value for the vertical scroll bar equals two. Our program uses these values to select the proper scrolling parameters.

The last line of the control resource definition specifies the initial settings for the current, minimum, and maximum values of the control. For our program, we choose a current value of zero, a minimum value of

Figure 6-9. Initial Placement of Scroll Bars



zero, and a maximum value of 500 for both controls. These values control the displacement (in pixels) of the picture relative to the window.

Drawing the Picture

The procedure "PenShapes" draws a diagram that illustrates Quickdraw's pen size attribute. It shows an array of pen sizes. However, it could easily display some other diagram or document. The procedure is independent of window management. It simply uses QuickDraw routines to make a picture. The exact appearance of that picture (how much is displayed and where) depends on the window's size and position on the screen.

The "PenShapes" procedure has four INTEGER parameters. The first two specify the horizontal and vertical displacement for scrolling the diagram. The second two specify the number of columns and rows in the display array. In this program, we display 400 pen sizes in a 20 by 20 array by passing a value of 20 for the last two parameters when this procedure is called.

Titling the Picture

The procedure erases the screen, then draws a fancy title at the top of the illustration. We call several QuickDraw routines to control the fanciness of the title.

We call "TextFont" to set the font. In this case, we use the constant "Athens" to specify font number seven. Table 6-2 lists other choices. Not all choices may be available on the disk you use.

We then call "TextFace" to set the style. The parameter for the "TextFace" routine is a Pascal set. More precisely, it is of type "Style", defined by the following Pascal type declarations:

```
StyleItem = (bold, italic, underline, outline,
             shadow, condense, extend);

Style = SET OF StyleItem;
```

In our program, we select "italic", "outline", and "shadow", passing them in Pascal's square bracket set notation.

Next, we call "TextSize" to set the text size to 24 points. The text size is the vertical distance between lines of text — about 72 points per inch; thus, our title is contained within a height of about one-third inch. In general, any integer size can be specified. However, the results look best when the text corresponds to an existing size for the selected font. The next best results occur when the text size is an even multiple of an existing

size. The Font Manager has routines to determine the existing fonts and sizes. Applications such as "Font Mover" also allow you to specify the sizes of the fonts on your disk.

The title is positioned with a "MoveTo" command. The position depends on the displacement "vector" (x,y) specified by the first two parameters of the "PenShapes" routine (see Figure 6-10). In our program, this displacement is controlled by the horizontal and vertical scroll bars.

To place the picture in the correct "scrolling" position, we subtract the displacement vector from the normal position. We center the title over the diagram, using a formula similar to the one in the "Title" procedure (see Chapter 5). We use "StringWidth" to size our title and "DrawString" to draw the title.

Now let's draw the pens — a study in relative motions (see Figure 6-11). First, we move to an absolute location within the window. This location is determined by subtracting the displacement vector (x,y) from the location of the first pen shape (in local coordinates). Everything in the diagram is drawn relative to this point.

The pen shapes are drawn within a double FOR loop that indexes all the rows and columns of the display. At the core of this double loop are three statements:

```
PenSize(I, J);
Line(0, 0);
Move(isize, 0);
```

Table 6-2. Fonts

Number	*Name*
0	systemFont (normally Chicago)
1	applFont
2	NewYork
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
8	SanFran
9	Toronto

These statements set the pen size, draw a single image of the pen, and move to the next horizontal position of the pen. Here, the constant "isize" specifies the size of the imaginary box occupied by each pen image, therefore the size of the relative move between boxes.

After each row follows the statement:

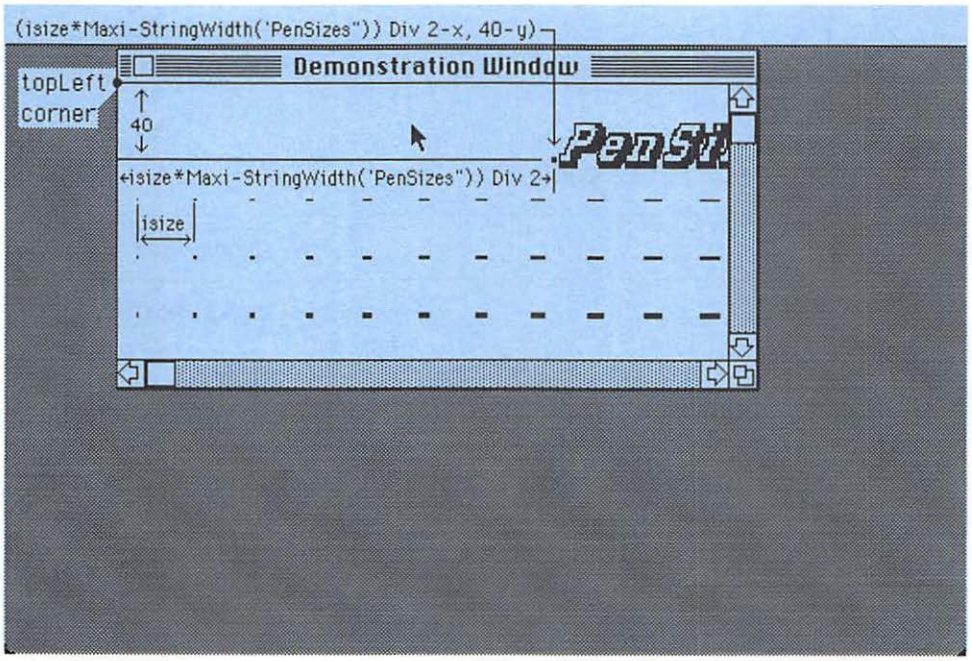
```
Move(-isize*MaxI, isize);
```

to move to the beginning of the next row. After the entire array is drawn, the "PenNormal" routine resets the attributes of the pen to normal and exits our procedure. If we do not call "PenNormal", lines around the scroll bars become too fat and cover the grow box and scroll bars.

Showing the Picture

The "ShowPict" routine "mounts" the diagram on the window. It interfaces the drawing procedure to the Window Manager's routines. Its only parameter is a window pointer, which points to the window that the picture is drawn in.

Figure 6-10. Positioning the Title



The procedure begins by calling the "ClipRect" routine to set the current limits of the clipping bounds region "clipBnds" (see Figure 6-12). As discussed in Chapter 4, QuickDraw automatically clips anything it draws to this grafPort region.

Clipping is needed because the full illustration is usually larger than the window area in which it is displayed. If the illustration is not clipped to this display area, then it spills into the scroll bars or beyond the window.

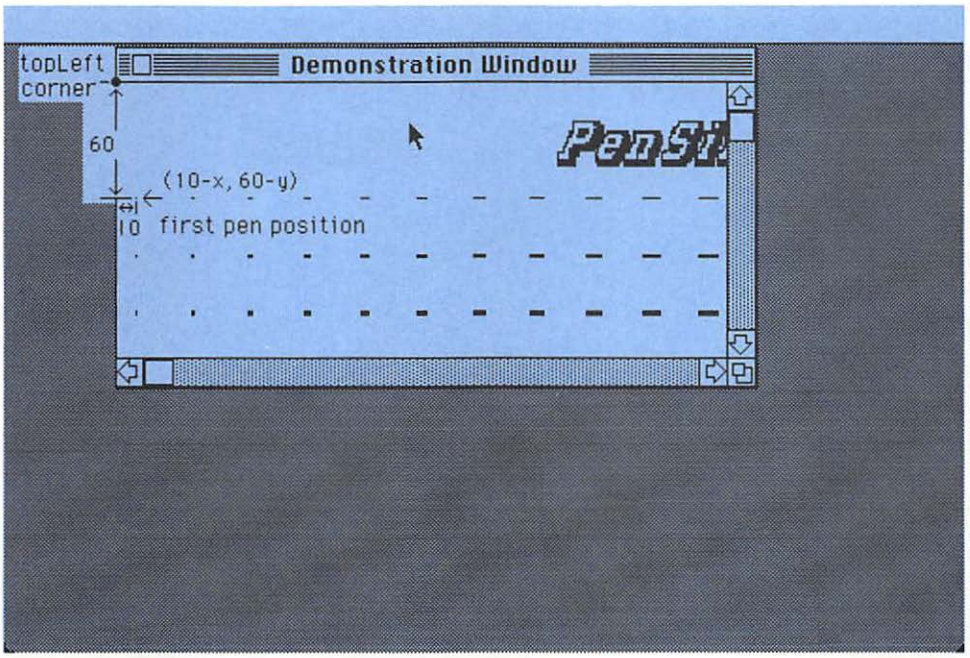
Since the window can "grow", this area changes size under the user's control and must be adjusted each time we draw the picture.

The limits of the clip region are computed by taking the window's port rectangle minus 15 pixels on the right and bottom for the scroll bars and grow box. This places the region in the center of the window, below the title bar, and to the left and above the scroll bars and grow box.

A WITH statement around this calculation simplifies the formula, allowing direct use of the various fields of theWindow^.portRect. We use the QuickDraw routine to set the clipping.

The next statement calls our "PenShapes" procedure, but only if the current window (given by the window pointer "theWindow") is our drawing window (given by the window pointer "myWindow"). In general, we

Figure 6-11. Relative Position of the Pens



make such tests to avoid drawing in the wrong window, such as one belonging to a desk accessory.

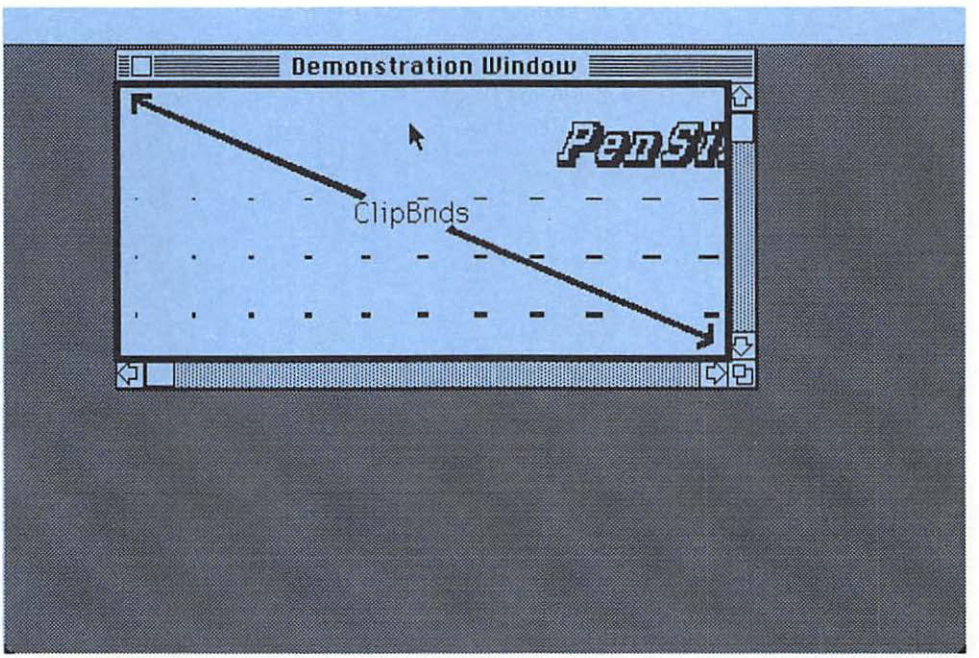
Of course, we don't have any desk accessories in this program, but the current window could be "empty". This happens when you select a point outside a window. In this case, the window pointer "theWindow" is NIL (zero value). Drawing under such circumstances could cause a very strange crash.

When dealing with several windows, we can use a CASE statement to branch to a procedure to draw the picture belonging to a given window.

When we call "Penshapes", we pass the scrolling displacement in the first two parameters. Using the "GetCtlValue" function, we fetch the horizontal displacement from the horizontal scroll bar control and the vertical displacement from the vertical scroll bar control. Thus, we use the control's own value field to store these essential quantities. We see the advantages of this later.

Before returning from this procedure, we reset the clip region to "drawRect". This ensures that the scroll bars and grow box operate properly. Otherwise, they might not update in response to the user's mouse commands.

Figure 6-12. The Clip Bounds



Growing Windows

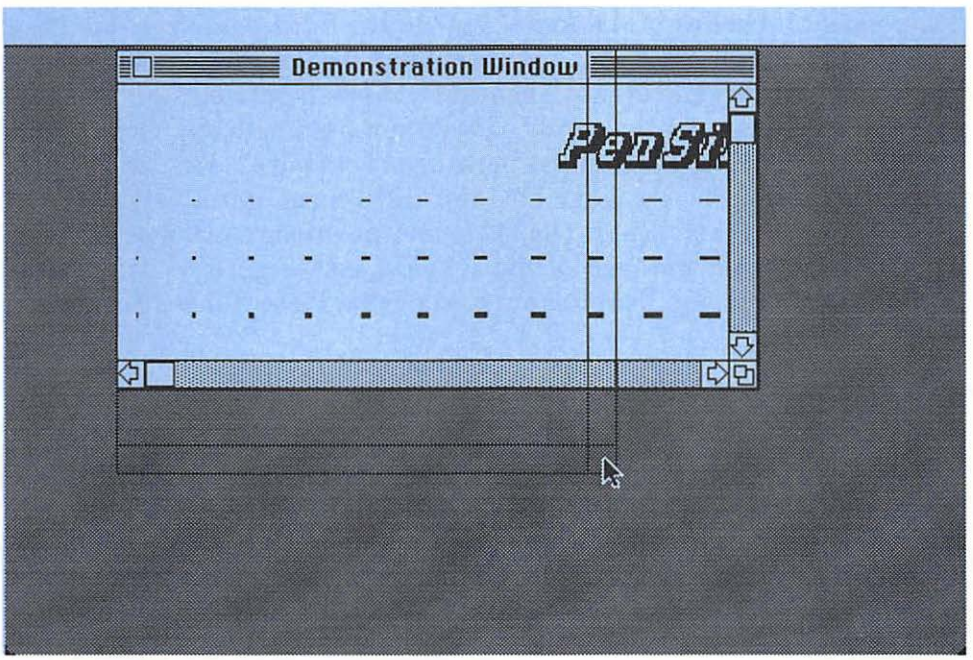
The next procedure, "WindowGrow", manages the resizing of the window in response to dragging the grow box (see Figure 6-13). This routine is called from the main program when the mouse is pressed in the grow box.

The procedure has two parameters: "theWindow", which is a window pointer, and "thePt", which is a point. The procedure "grows" "theWindow" starting with the cursor at "thePt".

The procedure has two variables: a long integer "WSize" and a point "S". Both hold the window's size, but differently.

The "WindowGrow" procedure first invokes the Window Manager's "GrowWindow" function to track the motions of the grow box and return the final window size when the mouse button is released. "GrowWindow" has three parameters: a window pointer, a point, and a rectangle. The first two parameters are the same as those described for our "WindowGrow" procedure, so we pass "theWindow" and "thePt" along from our procedure to this routine. The third parameter specifies the minimum and maximum allowable sizes for the window. We pass "SizeBnds" in this slot. The exact values are specified in the main program along with limits

Figure 6-13. Dragging the Grow Box



for other rectangles. "GrowWindow" returns a long integer that specifies the size in a "packed" form.

We grab the value returned from "GrowWindow" in the long integer "WSize". However, if the grow box is not moved but merely clicked, this returned value is zero. If WSize is zero, we should do nothing else: the size is wrong (zero!), and the scroll bars cannot move properly in response to no amount.

If WSize is zero (no motion), we call the Pascal "Exit" procedure to exit from the routine. If the window is to be resized, we continue. (We could instead surround the last part of this routine with an IF...THEN statement, but that increases the complexity of the program's structure, adding another level of indentation.)

First, we use "SetPt" with the "loWord" and "hiWord" functions to convert the window's size from its "packed" form in the long integer "WSize" to "S", which is a point. This provides easier access to its individual horizontal and vertical components.

Next, we call "SizeWindow" to move the window frame. This routine has four parameters: a window pointer to the affected window, an integer to specify the new horizontal size, an integer to specify the new vertical size, and a Boolean to specify whether the system should generate an update event after the window is resized by this command. We pass "theWindow" to the first parameter. We pass the horizontal component of the size in the form "S.h" to the second parameter. We pass the vertical component of the size in the form "S.v" to the third parameter. We pass "true" to the fourth parameter, indicating that we want the Window Manager to generate update events when the window is resized.

The growIcon and the scroll bars do not automatically move when the window is resized. We must move them ourselves. We call "DrawGrowIcon" to redraw the grow box and call "SizeControl" and "MoveControl" for each scroll bar. The sizes and positions for the scroll bars depend directly on the size of the window (see Figure 6-14). Closely examine the figure and the program to see what these dimensions are.

The Scroll Routine

The next procedure, "ScrAction", is unusual: it involves a unique cooperation between the Macintosh's built-in software and an applications program. It is never directly called by an applications program. Instead, it is an *action routine* called by the Control Manager's *tracking* routine to repeatedly perform a specified action during tracking. This occurs when our program calls the tracking routine, which in turn calls the action procedure.

When we track the scroll bars in our program, we want to repeatedly scroll the picture as the user presses the up or down buttons or the page up or down controls. Yet, we want the Control Manager to perform highlighting and other tracking operations.

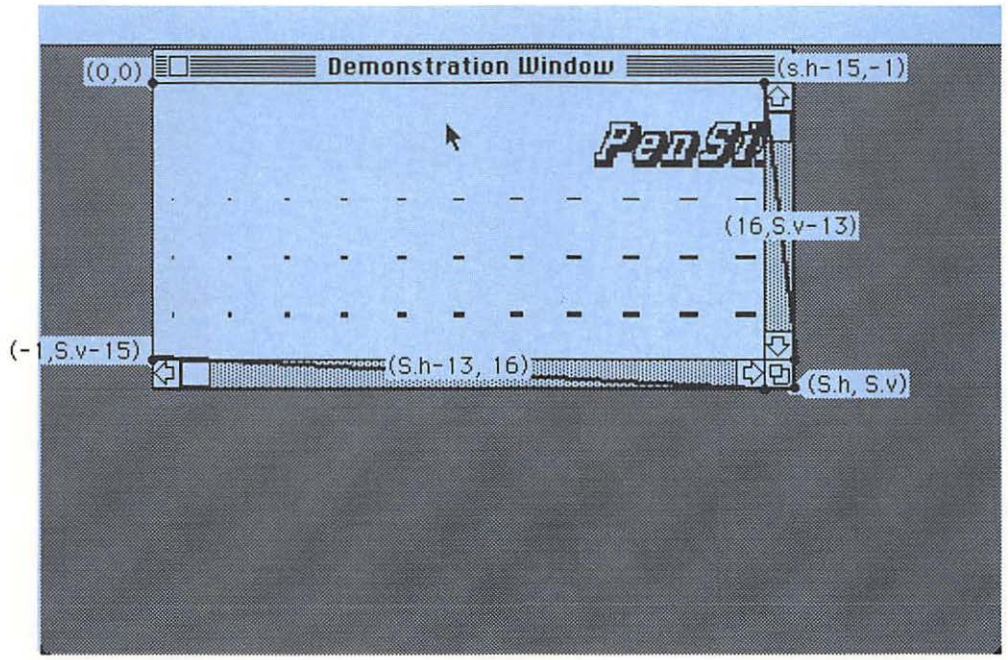
Our control action routine has two parameters: a control handle to specify the control being tracked and a part code to specify a particular part of that control.

The parts for scroll bars have codes, specified by the following Pascal constants statements:

```
inUpButton   = 20;
inDownButton = 21;
inPageUp     = 22;
inPageDown   = 23;
inThumb      = 129;
```

The first four parts require special action; the last, "inThumb", does not. The difference is that the first four parts are "buttonlike" and signal the program that the user wants to do something; the last part is an

Figure 6-14. Relative Size and Positions of Scroll Bars



"indicator" that the Control Manager knows how to drag and update. In our case, we tell the Control Manager to use our action procedure only if the first four parts occur. If an action routine were used for the "inThumb" case, we would require a different syntax for our action routine — one with no parameters.

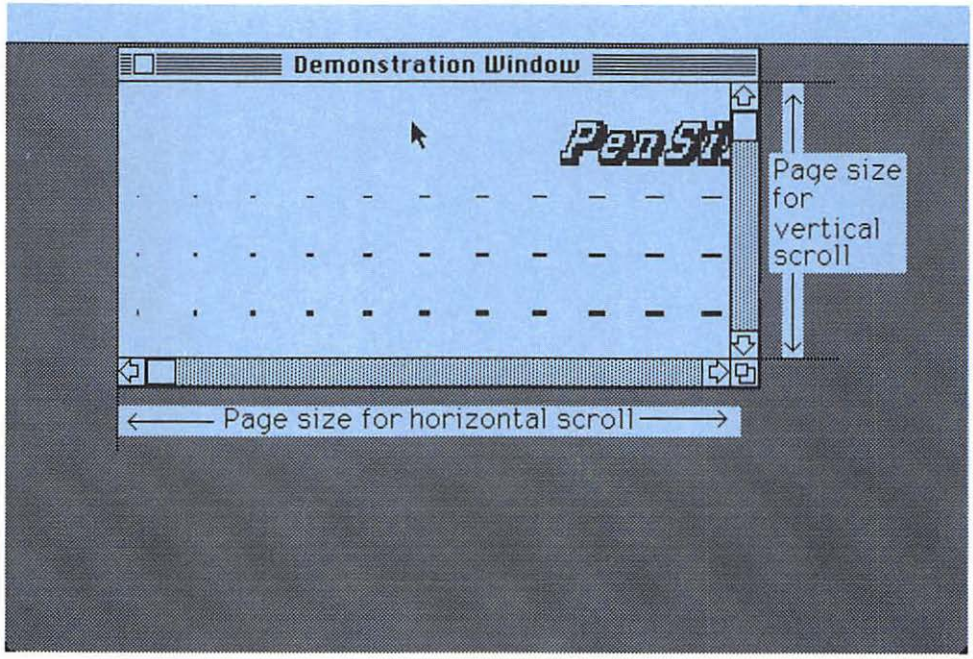
Now let's look at our action procedure. It has two local variables: "pagesize" and "delta", both integers. These help to determine the amount and direction of scrolling.

The procedure begins by setting the "pagesize". This is the distance that we scroll if the user selects "page up" or "page down" (see Figure 6-15).

The value of "pagesize" depends on the size of the current port rectangle as well as the direction in which we scroll. The entire statement is surrounded by a "WITH thePort^.thePort DO" statement so that we can directly use the "right", "left", "bottom", and "top" fields of this rectangle.

We use the reference values in the horizontal and vertical scroll bar controls to drive a CASE statement. A value of one indicates the horizontal

Figure 6-15. Page Size



scroll bar. A value of two indicates the vertical scroll bar. The formula for the horizontal direction is:

```
pagesize := right - left - 16;
```

The formula for the vertical direction is:

```
pagesize := bottom - top - 16;
```

We must subtract 16 in each formula due to the space occupied by the scroll bars.

We immediately exit the procedure if neither case occurs.

We then see which part of the control is selected and set "delta" accordingly. If none of the four parts is selected, we immediately exit the procedure.

Now we use "GetCtlValue" and "SetCtlValue" to increment the control's value by "delta". We then call "ShowPict" to display our picture in its new position. That's all there is to scrolling: change the displacement and redraw the picture. However, the procedure can be refined. For example, "ScrollRect" may shift only part of the picture, and the update region can be set to redraw only newly visible parts.

Test the scrolling controls to see how all this works.

Managing Window Scrolling

The routine "WindowScroll" manages scrolling in our program. It is called from the main program when the mouse is pressed in the content region of the window.

The procedure has two parameters: a window pointer that points to the window we want to scroll, and a point that is the current mouse position.

The procedure has a local variable, "theCtl", a control handle used to select one of the two scroll controls.

We begin the procedure by calling the QuickDraw "SetPort" routine to make the grafPort of the specified window into the current grafPort. Next, we call the QuickDraw "GlobaltoLocal" procedure to convert the mouse position from global to local coordinates. Once a window is selected, its local coordinates are used for all drawing and all controls.

Now we find and track the controls. First, we call the Control Manager's "FindControl" function to see if the mouse is pressed in one of the two controls.

The Control Manager's "FindControl" has three parameters: a point that specifies the current mouse position in the window's local coordinates, a window pointer that points to the desired window, and a control handle that is passed by reference. If the mouse is pressed in a control, the function returns a handle to it in the last parameter. The function also returns the part code as the return value of the function. If the mouse is not pressed in a currently active control, it returns a zero as the part code and NIL for a control handle. The special feature called "hilite code" is not used.

We feed the part code from "FindControl" into a CASE statement. Each case uses the "TrackControl" function to track the control; that is, perform normal highlighting as the user moves the mouse with the mouse button down.

The "TrackControl" function has three parameters: a control handle to the selected control, a point that is the current position of the mouse in local coordinates, and a pointer to the programmer's action procedure (if any). The "TrackControl" funtion returns the part code once the mouse button is released. If the mouse is released in a part other than where pressed, this part code is zero.

If the part code from "FindControl" indicates one of the "buttonlike" controls, we call "TrackControl", passing the address of our "ScrollAction" procedure to update the control's value and the picture during tracking. In this case, the "TrackControl" function is surrounded by a "dummy" IF..THEN statement, since we don't want to waste a variable to pick up its returned part code. This also indicates that nothing is required after tracking.

If the part code from "FindControl" indicates the thumb indicator, we call "TrackControl", passing a pointer value of NIL for the action routine. Again, we place "TrackControl" in an IF..THEN statement. However, in this case, we call our "ShowPict" routine if the returned part code is nonzero; that is, the picture is redrawn only after all the tracking is completed.

Updating and Activating a Window

Our last procedure, "WindowUpdate", controls the updating and activation of our display window. Updating occurs when a window is resized or dragged so that parts formerly hidden become visible. These need to be redrawn, or at least specified, by the applications program. Activation occurs when a window is first created or selected. With several windows, both *activate* and *deactivate* events occur as different windows are se-

lected. In our case, the activate event occurs only once as a result of the "GetNewWindow" command.

Our "WindowUpdate" procedure has one parameter: a window pointer to the window that needs attention.

We begin the procedure by calling the QuickDraw "SetPort" routine to make the grafPort of the specified window into the current grafPort. Next, we call the Window Manager's "BeginUpdate" routine to begin the update process. This routine replaces the window's "visRgn" with the intersection of the window's visRgn and update region. The picture is thus restricted to only those areas of the window needing attention. The window's update region is then emptied, assuming that updating is complete for that particular window.

The next two lines are just for fun and should not be placed in a real application. The first inverts the update region, the second beeps for a time. This gives a chance to see the areas of the screen being updated.

Now we update. Be aware that three separate parts of the system cooperate in the updating process. The Control Manager's "DrawControls" is called to draw the scrolling controls. The Window Manager's "DrawGrowIcon" is called to draw the grow box. The "ShowPict" procedure is called to draw the picture.

Finally, we call "EndUpdate" to restore the orginal visRgn of the window's grafPort.

The Main Program

The main program controls the flow of events. Like the example program in Chapter 5, it has an initialization section and a main loop.

Initializing Section

In the initialization section, we call our "SetUp" procedure to initialize QuickDraw and the various managers. Next, we initialize the three rectangles, "drawRect", "sizeBnds", and "dragBnds", which define the delimiting parameters to draw, resize, and drag. In this program, these limits are gathered in one place for the convenience of the programmer who maintains this program.

The main loop is a REPEAT...UNTIL loop. As before, the Boolean variable "done" in the UNTIL clause controls the loop. Before the loop, it is set to false. In the loop, the user can set it to true. That happens in this program if the user selects the window's "goAway" box.

As in the Chapter 5 program, the loop begins by calling the "GetNextEvent" function in an IF...THEN statement. The THEN part is exe-

cuted only if the Boolean from "GetNextEvent" is true, indicating that our program is to handle the event.

At the heart of the loop is a CASE statement driven by the "GetNextEvent" function. In this program, we want to deal only with "mouseDown", "update", and "activate" events. Of course, you can add other cases to expand the capabilities of the program. Within the "mouseDown" case is another CASE statement driven by the "FindWindow" function to determine which part of the window the mouse was in when pressed. It also finds out which window it was in. Of course, in our program there is only one window to find.

The "FindWindow" function has two parameters: a point to show where the mouse cursor is or was; and a window pointer that, upon return, points to the mouse's window. The last parameter is passed by reference. The function returns an integer with a "part code" for the window. The following Pascal statements give the standard window part codes:

```
inDesk = 0;
inMenuBar = 1;
inSysWindow = 2;
inContent = 3;
inDrag = 4;
inGrow = 5;
inGoAway = 6;
```

In this program, we use only the last four statements: "inContent", "inDrag", "inGrow", and "inGoAway".

If the mouse is pressed in the content region, we branch to the "inContent" case, where we call our "WindowScroll" procedure. It verifies that the mouse position is in one of the scroll controls. If so, it provides the appropriate tracking and scrolling.

If the mouse is pressed in the drag region, which for our window is the title bar (except for the goAway box), we branch to the "inDrag" case, where we call the Window Manager's "DragWindow" routine. This routine handles the entire window dragging process except for updating. The Window Manager generates an update event when parts of the window's contents are dragged into view from outside the viewing screen.

The "DragWindow" routine has three parameters: a window pointer that points to the affected window, a point that contains the mouse position in global coordinates when the button is pressed, and a rectangle that specifies the limits in global coordinates so that you can drag the top left corner of the content region of the window. In our program, the "dragBnds" is passed as this last parameter. Earlier, it was set to a rectangle slightly

smaller than the active part of the screen so that the window is never dragged completely out of sight.

If the mouse is pressed in the grow box, we branch to the "inGrow" case, where we call our "WindowGrow" procedure. This routine resizes our window. It usually generates update events, which we handle with the "update" case.

If the mouse is pressed in the goAway box, we branch to the "inGoAway" case, where we call the Window Manager's "TrackGoAway" function. This function does the proper highlighting actions for tracking the cursor, which in this case is drawing little star-shaped lines — or "highlights" — around the goAway box until the user releases the mouse button. It then returns with a Boolean that is true if the mouse is still in the goAway region and false if not. In our case, we assign this result to our Boolean variable "done".

These are all the cases for "mouseDown". For "update" and "activateEvt", we call our "WindowActivate" routine. Again, this updates (redraws) just those portions of the content region of the window that need updating. In this program, we can use the same procedure to update and activate windows. For update and activate events, the ".message" field of the event record contains a numerical value that is the address of a pointer to the given window. We pass this pointer value in the following form:

```
POINTER(theEvt.message)
```

to our "WindowUpdate" routine.

That's all there is to the loop. It continues until the go away condition is met by the user.

Summary

In this chapter, we have studied the Window Manager and the Control Manager, which allow us to control a single window on the screen. We have seen how to scroll, drag, resize, and update a window, and how to make it go away. In Chapter 7, we introduce those concepts that allow us to handle several windows at once.

This chapter covered the following ROM routines:

WM-InitWindows

WM-GetNewWindow

CM-GetNewControl

QD-TextFont
QD-TextFace
QD-TextSize
QD-PenSize
QD-Move
QD-PenNormal
CM-GetCtlValue
WM-GrowWindow
WM-SizeWindow
CM-SizeControl
CM-MoveControl
WM-DrawGrowIcon
CM-GetCRefCon
CM-SetCtlValue
QD-GlobaltoLocal
CM-FindControl
CM-TrackControl
WM-BeginUpdate
OU-SysBeep
CM-DrawControls
WM-EndUpdate
WM-FindWindow
WM-DragWindow
WM-TrackGoAway

7

Overlapping Windows

This chapter covers the following new concepts:

- **Pictures**
- **Polygons**
- **String Resource**
- **Overlapping Windows**
- **Hiding and Showing Windows**
- **Window Selection and Highlighting**
- **Window Updating**

In this chapter we describe how to manage multiple overlapping windows. We also introduce more advanced techniques in picture making and window management, including *pictures*, *polygons*, and fine scrolling. We begin with short descriptions of *pictures* and *polygons*, then describe a programming example that illustrates how to manage several windows at once. We fill one window with an image drawn with a *picture* and fill another window with a series of images drawn with *polygons*.

Pictures

A *picture* is a list of basic QuickDraw drawing commands in a compressed format in which each drawing command is encoded as a single-command byte followed by data bytes. This is more efficient than using the command's full name or trap code. As we see later, frequently used commands have shortened versions for easier access.

A picture is a way to handle graphics information to be displayed in a Macintosh window. However, uses of pictures include storage of graphics information on disk and transmission of graphics information to a remote device such as an intelligent printer. The compressed format is useful in these applications because it saves memory space, transmission time, and execution time.

In this section, we briefly review this format so that you understand the picture-drawing facilities of Quickdraw. For example, you see how quantities that determine the size and shape of objects in the picture are frozen as constants when the picture is made. This is valuable when planning or debugging your own graphics programs.

As of this writing, Apple has not documented this format, which makes it subject to change. The routines that generate and interpret these commands are in ROM, so any change is unlikely, although possible, since new RAM routines can be substituted for the original ROM routines.

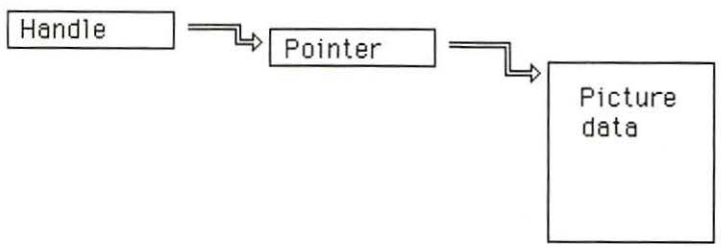
A picture is accessed through a *picture handle* that is defined through the following Pascal data statements:

```
PicHandle = ^PicPtr;
PicPtr    = ^Picture;
Picture   = RECORD
              picSize:  INTEGER;
              picFrame: Rect;
              {picture definition data}
                END;
```

That is, a "PicHandle" is a pointer to a "PicPtr", which in turn points to a "Picture" (see Figure 7-1). Recall that this double pointer system is required if the picture is to be properly handled as a dynamic variable.

Given the above declaration, we see that "Picture" is a record consisting of two fields that Pascal knows about, plus data that only QuickDraw

Figure 7-1. Picture Handles, Pointers, and Data



knows about. The Pascal field ".picSize" gives the number of data bytes in the picture. This information is needed by the Memory Manager. The field ".picFrame" is a rectangle that encloses the picture.

The picture definition data follows the second field. Picture-drawing commands are stored in this data as sequences of bytes, like a kind of machine language. In this case, the machine language consists of graphics commands. These commands are called picture definition code rather than central processor commands. Think of the picture-drawing process as running a graphics "processor" that executes these picture-drawing commands. Of course, this particular graphics processor is constructed of software. But imagine a hardware processor, perhaps in some peripheral equipment, that reads and executes such graphics "programs".

Each picture-drawing command begins with a command byte, acting like a machine language operation code. Following each command byte are several data bytes. These data bytes contain "literal" values needed by QuickDraw to draw the picture. For example, the command to change the pen size to (2, 3) is encoded in hexadecimal as follows:

```
07, 0003, 0002
```

The "07" is the command byte for pen size, "0003" is an integer constant that specifies the vertical size of the pen, and "0002" is an integer constant that specifies the horizontal size of the pen. Notice that the vertical component is stored first, corresponding to the way these quantities are stored internally.

There are a variety of line-drawing commands. For example, the command byte with hexadecimal value $20 followed by integers y1, x1, y2, x2 draws a line from (x1,y1) to (x2,y2), and the command byte $21 followed by integers y, x draws a line from the previous current position to the point (x,y). Also, a command for "short" lines uses byte-sized vertical and horizontal displacements. This last command uses only three bytes of picture definition code, therefore runs faster and requires less storage.

The commands for drawing text specify a location (absolute or relative) and a literal string. For example, a command byte $28 followed by integers y and x and a string moves the pen to the position (x,y) and draws the string there.

It is important to understand that all expressions are evaluated when the picture definition is created and that only the resulting constant or literal values are stored in the picture definition data. This means that once the picture is created, you cannot effectively control parameters in

your program to determine the placement of lines, the size and spacing of rectangles or text strings, and the patterns to fill shapes.

Note that not every QuickDraw command is immediately converted to a corresponding picture definition command. For example, "Move" and "MoveTo" are not translated until something is drawn, such as a line or a string.

Polygons

Polygons are figures formed by a series of line segments (see Figure 7-2). Once a polygon is defined, it can be outlined, filled, erased, or inverted, just like rectangles, ovals, and regions in previous chapters.

The "Stars" procedure of our example program, discussed later, illustrates how polygons are defined and drawn. You may want to look at it before proceeding with this discussion.

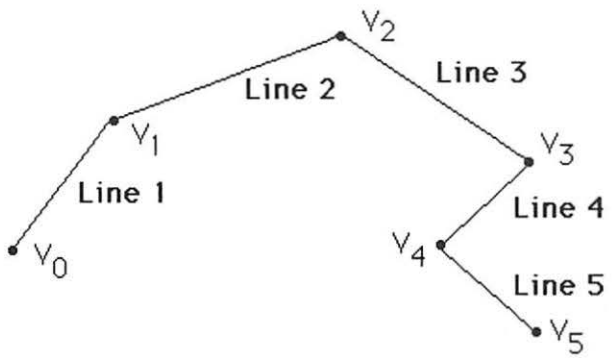
Structure of Polygons

We begin with the internal structure of polygons. Normally, the programmer refers to a polygon only by its handle, not needing to know the internal structure. But to truly understand polygons, we must look at this structure.

A polygon handle is a variable of type "polyHandle", defined by the following declaration:

```
PolyHandle = ^PolyPtr;
PolyPtr    = ^Polygon;
```

Figure 7-2. A Polygon



```
Polygon     = RECORD
   polySize:     INTEGER;
   polyBBox:     Rect;
   polyPoints:  ARRAY [0..0] OF Point;
END;
```

That is, a "PolyHandle" is a pointer to a "PolyPtr", which points to a "Polygon". A "Polygon" is where the data is stored.

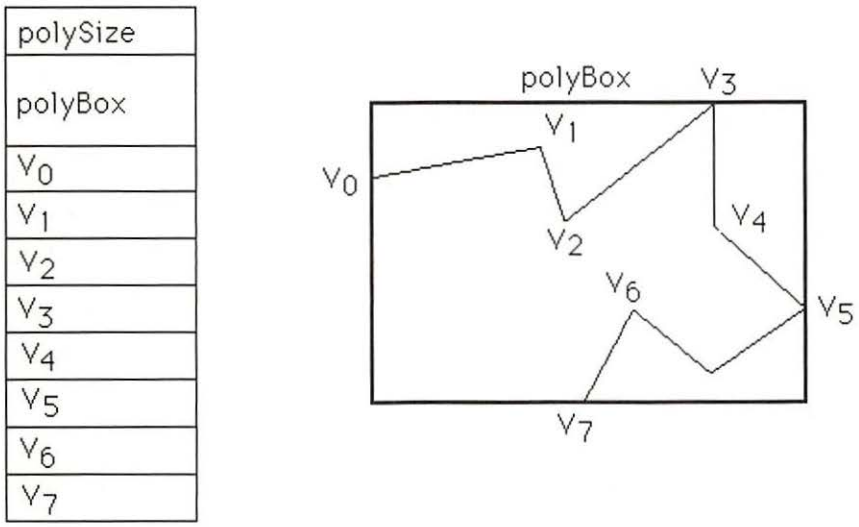
A *polygon* is a record structure with three fields (see Figure 7-3). The first field, ".polySize", is an integer that specifies the number of data bytes stored in the polygon. This is required for memory management. The second field, ".PolyBBox", is a rectangle that bounds the polygon. In our "star" example, we see how this ".polyBBox" increases the performance of our drawing.

The third field, ".polyPoints", is an array of points that specifies the vertices of the polygon. Later, we explain how to load values into these vertices.

Defining Polygons

Polygons are defined using the "OpenPoly" function and the "ClosePoly" procedure. The "OpenPoly" function returns a handle to a newly created

Figure 7-3. Internal Structure of Polygons



polygon, turns off drawing to the screen, and causes subsequent line-drawing commands to accumulate as part of the polygon.

Once a polygon is open, you can define the polygon with "Line" and "LineTo" commands (and perhaps an initial "Move" or "MoveTo"). The first "Line" or "LineTo" command places the two end points of the line in the ".polyPoint" array. The first point is the current position before the first line is drawn. The second point is the current position after the "line" command. Each subsequent line-drawing command places the new position in the ".polyPoint" array. Thus, the number of vertices is always one more than the number of lines drawn.

The polygon is closed by the "ClosePoly" command. This computes a rectangle that fits around the polygon, storing it in the ".polyBBox" field of the polygon, and shows the pen again.

Drawing Polygons

Once a polygon is defined, you can use the commands "FramePoly", "FillPoly", "PaintPoly", "ErasePoly", and "InvertPoly" to draw with it. Each of these commands is invoked with the polygon handle as a parameter.

You can use the "OffsetPoly" routine to move a polygon around, drawing anywhere on the screen. We do this in our "stars" procedure.

The Example Program

The example program in this chapter demonstrates how to manage several windows at once. The windows can be moved and resized so that they overlap or even hide each other. The Window Manager takes care of all such details if we do our part.

When the program signs on, only one window is visible (see Figure 7-4). It is called the control window, and its sole function is to control other windows. It has the form of a rounded rectangle and contains two *control* buttons. A button labeled "graphics" makes a window containing graphics appear. The button labeled "text" makes a window containing text appear (see Figure 7-5). The control window has a goAway box that, when pressed, terminates the program.

The text window and the graphics windows are called display windows because they display graphics or textual information. Both have vertical and horizontal scroll bars that allow scrolling by thumb, page, and button controls. They also have grow boxes for resizing and goAway boxes to make them disappear.

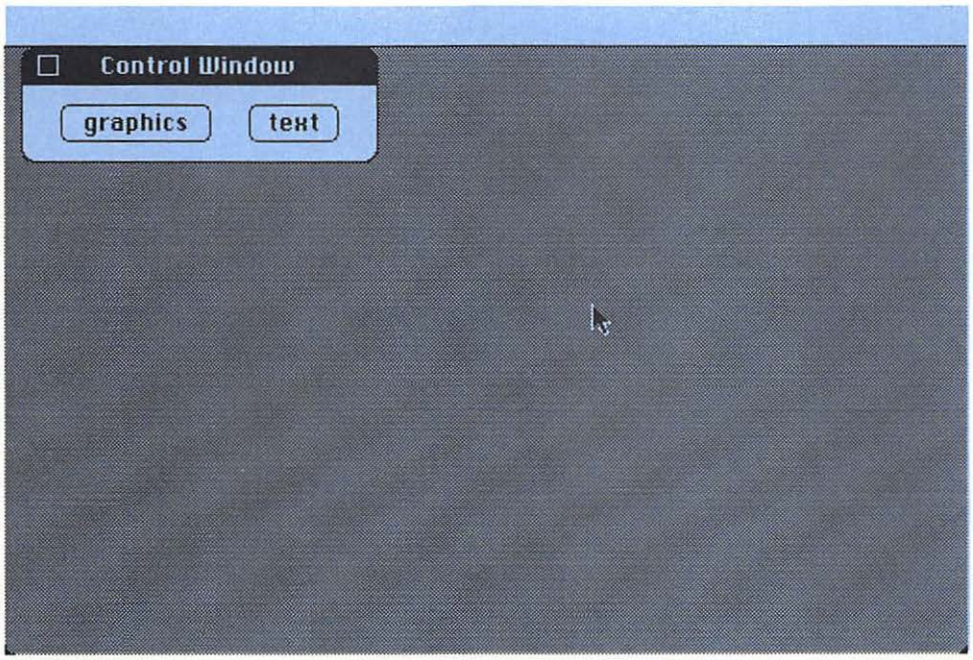
The graphics window displays an array of stars, illustrating absolute and relative "move" and "line" commands and polygon drawing.

The text window displays some text that describes the program. This window illustrates string resources and pictures.

Now let's look at the program. It is longer than our previous programs, but it is divided into easy-to-understand segments.

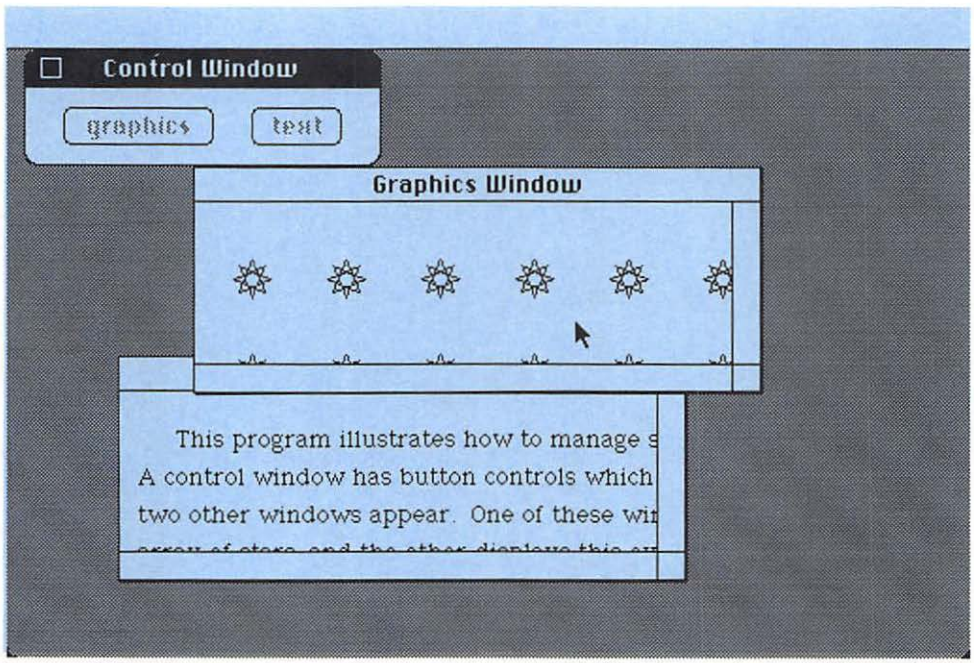
```
PROGRAM Windows;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf;

   CONST
      isize = 4;

   VAR
      done: boolean;
      theEvt: eventRecord;
      CtlWindow, theWindow: WindowPtr;
```

Figure 7-4. The Control Window



```
    URgn: RgnHandle;
    dragBnds, sizeBnds: Rect;
    Pict:           ARRAY [1..2] OF PicHandle;
    picBnds:        ARRAY [1..2] OF Rect;
    CtlButton:      ARRAY [1..2] OF ControlHandle;
    hsbar, vsbar:   ARRAY [1..2] OF ControlHandle;

PROCEDURE SetUpSys;
    BEGIN
        InitGraf(@thePort);
        InitFonts;
        InitWindows;
        FlushEvents(everyEvent,0);
        SetEventMask(everyEvent);
        InitCursor;
        URgn := NewRgn;

        SetRect(picBnds[1],     0,      0,    512,    342);
        SetRect(picBnds[2],     0,      0,    512,    342);
        SetRect(sizeBnds,      50,     50,    512,    342);
        SetRect(dragBnds,       4,     24,    508,    338);
```

Figure 7-5. The Graphics and Text Windows



```
      done := false;
   END;

PROCEDURE Stars(VRgn : RgnHandle);
   CONST
      maxI = 10;
      maxJ = 10;
   VAR
      I, J : INTEGER;
      Star : PolyHandle;
   BEGIN
      Moveto(10, 0);
      Star := OpenPoly;
         Line(-17, -7);
         Line(7, 17);
         Line(7, -17);
         Line(-17, 7);
         Line(17, 7);
         Line(-7, -17);
         Line(-7, 17);
         Line(17, -7);
      ClosePoly;
      OffsetPoly(Star,30,40);
      FOR I := 1 to MaxI DO BEGIN
         FOR J := 1 to MaxJ DO BEGIN
            IF RectInRgn(Star^^.polyBBox,VRgn) THEN FramePoly(Star);
            OffsetPoly(Star,50,0);
         END;
         OffsetPoly(Star,-50*MaxI,50);
      END;
   END;

PROCEDURE Explain;
   VAR
      theText : Handle;
      IPtr : ^INTEGER;
      S : StringPtr;
      i: INTEGER;
   BEGIN
      theText := GetResource('STR#',256);
      HLock(theText);
         IPtr := POINTER(ORD(theText^));  {point to num of strings}
         S := POINTER(ORD(theText^) + 2); {point to first string}
         FOR i := 1 TO IPtr^ DO BEGIN
            MoveTo(10,10+20*i);                {beginning of line}
            DrawString(S^);                    {draw the string}
            S := POINTER(ORD(S)+LENGTH(S^)+1); {point to next string}
         END;
      HUnLock(theText);
   END;
```

```
PROCEDURE SetUpWindows;
   BEGIN
      {set up control window}
      CtlWindow := GetNewWindow(257,NIL,POINTER(-1));
      CtlButton[1] := GetNewControl(257, CtlWindow);
      CtlButton[2] := GetNewControl(258, CtlWindow);

      {set up graphics window}

      theWindow := GetNewWindow(258,NIL,NIL);
      SetCRefCon(CtlButton[1],ORD(theWindow));
      hsbar[1] := GetNewControl(259,theWindow);
      vsbar[1] := GetNewControl(260,theWindow);

      {set up text window}
      theWindow := GetNewWindow(259,NIL,NIL);
      SetCRefCon(CtlButton[2],ORD(theWindow));
      hsbar[2] := GetNewControl(261,theWindow);
      vsbar[2] := GetNewControl(262,theWindow);
      SetPort(theWindow);
      ClipRect(picBnds[2]);
      Pict[2]:= OpenPicture(picBnds[2]);
         Explain;
      ClosePicture;
   END;

PROCEDURE UpdatePic(i : INTEGER; URgn : RgnHandle);
   VAR
      S: Point;
   BEGIN
      If i = 0 THEN Exit(UpdatePic);
      SetPt(S,GetCtlValue(hsbar[i]),GetCtlValue(vsbar[i]));
      SetOrigin(S.h,S.v);
      OffsetRgn(URgn,S.h,S.v);
      SetClip(URgn);
      EraseRgn(URgn);
      CASE i OF
         1: Stars(URgn);
         2: BEGIN
               Hlock(Handle(Pict[i]));
                  DrawPicture(Pict[i],picBnds[i]);
               HUnlock(Handle(Pict[i]));
            END;
         END;
      SetOrigin(0,0);
      ClipRect(thePort^.portRect);
   END;
```

```
PROCEDURE ScrAction(theCtl: ControlHandle; partCode: INTEGER);
   VAR
      pageSize, delta,  i : INTEGER;
      S, dS : Point;
      viewBnds : Rect;
   BEGIN
      i := GetWRefCon(theWindow);
      If i = 0 THEN Exit(ScrAction);

      WITH thePort^.portRect DO
         CASE GetCRefCon(theCtl) OF
            1: pagesize := right - left - 16;
            2: pagesize := bottom - top - 16;
            otherwise   Exit(ScrAction);
            END;
      CASE partCode OF
         inUpButton:   delta := -isize;
         inDownButton: delta := +isize;
         inPageUp:     delta := -pagesize;
         inPageDown:   delta := +pagesize;
         otherwise     Exit(ScrAction);
         END;

      SetPt(S,GetCtlValue(hsbar[i]),GetCtlValue(vsbar[i]));
      SetCtlValue(theCtl,GetCtlValue(theCtl)+delta);
      SetPt(dS,S.h-GetCtlValue(hsbar[i]),S.v-GetCtlValue(vsbar[i]));

      WITH thePort^.portRect DO
         SetRect(viewBnds,left,top,right-15,bottom-15);
      ScrollRect(viewBnds,dS.h,dS.v,URgn);
      UpdatePic(i,URgn);
   END;

PROCEDURE WindowControl(thePt: Point);
   VAR
      theCtl  : ControlHandle;
   BEGIN
      IF theWindow = FrontWindow THEN BEGIN
         SetPort(theWindow);
         GlobalToLocal(thePt);
         CASE FindControl(thePt,theWindow,theCtl) OF
            inButton:
               IF TrackControl(theCtl,thePt,NIL) <> 0 THEN BEGIN
                  HiliteControl(theCtl,255);
                  ShowWindow(POINTER(GetCRefCon(theCtl)));
               END;
            inUpButton, inDownButton, inPageUp, inPageDown:
               IF TrackControl(theCtl,thePt,@ScrAction) <> 0 THEN;
            inThumb:
```

```
                IF TrackControl(theCtl,thePt,NIL) <> 0 THEN BEGIN
                   WITH theWindow^.portRect DO
                      SetRectRgn(URgn,left,top,right-15,bottom-15);
                   UpdatePic(GetWRefCon(theWindow),URgn);
                END;
             END;
       END
       ELSE BEGIN
          SelectWindow(theWindow);
          DrawGrowIcon(theWindow);
          DrawControls(theWindow);
       END;
    END;

PROCEDURE WindowGrow(i: INTEGER);
    VAR
       Wsize : LONGINT;
       S : Point;
    BEGIN
       WSize := GrowWindow(theWindow,theEvt.where,sizeBnds);
       IF WSize = 0 THEN Exit(WindowGrow);

       SetPt(S,loWord(WSize),hiWord(WSize));
       SizeWindow(theWindow,S.h, S.v, true);
       SetPort(theWindow);
       ClipRect(thePort^.portRect);
       SizeControl(hsbar[i],S.h-13,     16);
       MoveControl(hsbar[i],    -1, S.v-15);
       SizeControl(vsbar[i],    16, S.v-13);
       MoveControl(vsbar[i],S.h-15,     -1);
    END;

PROCEDURE WindowGoAway(i: INTEGER);
    BEGIN
       IF TrackGoAway(theWindow,theEvt.where) THEN
          CASE i OF
             0: {the control window}
                done := TRUE;
             1, 2: {the display windows}
                BEGIN
                   HideWindow(theWindow);
                   HiliteControl(CtlButton[i],0);
                END;
             END;
    END;

PROCEDURE WindowUpdate(i : INTEGER);
    VAR
       growArea: Rect;
```

```
    BEGIN
       SetPort(theWindow);
       IF i <> 0 THEN BEGIN
          WITH thePort^.portRect DO
             SetRect(growArea,right-15,bottom-15,right,bottom);
          InvalRect(growArea);
          IF theWindow = FrontWindow THEN BEGIN
             ShowControl(hsbar[i]);
             ShowControl(vsbar[i]);
          END
          ELSE BEGIN
             HideControl(hsbar[i]);
             HideControl(vsbar[i]);
          END;
       END;
       BeginUpdate(theWindow);
          InvertRect(theWindow^.portRect);
          SysBeep(10);
          EraseRect(theWindow^.portRect);
          DrawGrowIcon(theWindow);
          DrawControls(theWindow);
          WITH theWindow^.portRect DO
             SetRectRgn(URgn,left,top,right-15,bottom-15);
          UpdatePic(i,URgn);
       EndUpdate(theWindow);
    END; {Update}

BEGIN {main program}
    SetUpSys;
    SetUpWindows;

    REPEAT
       IF GetNextEvent(everyEvent,theEvt) THEN
          CASE theEvt.what OF
             mouseDown:
                CASE FindWindow(theEvt.where,theWindow) OF
                   inDesk:
                      SelectWindow(CtlWindow);
                   inContent:
                      WindowControl(theEvt.where);
                   inDrag:
                      DragWindow(theWindow,theEvt.where,dragBnds);
                   inGrow:
                      WindowGrow(GetWRefCon(theWindow));
                   inGoAway:
                      WindowGoAway(GetWRefCon(theWindow));
                END; {FindWindow}
```

```
            UpdateEvt, ActivateEvt: {update window}
               BEGIN
                  theWindow := POINTER(theEvt.message);
                  WindowUpdate(GetWRefCon(theWindow));
               END;
         END; {Event}
   UNTIL done;
END.
```

Data Structures

The USES section is the same as before. The CONST section contains a global constant, "isize", to determine the displacement for fine scrolling when using the scroll bars. It is set to a value of four pixels per scroll. A smaller value yields finer, but slower, scrolling. A larger value yields larger, but faster, scrolling.

Global Variables

The VAR section contains declarations for several variables. The first two, "done" and "theEvt", are used in the example programs in Chapters 5 and 6 to help manage events. The next two variables are window pointers. The first window pointer, "CtlWindow", permanently points to the control window. The second window pointer, "theWindow", is a general-purpose window pointer that points to the window being selected or modified.

The global variable "URgn" helps to update windows. It is a region handle used repeatedly. Apple indicates that you should dispose of these handles after each use; but reusing the same handle has about the same effect.

The next two rectangles, "dragBnds" and "sizeBnds", set limits on dragging and sizing. These limits are explained in Chapter 6.

The last global variables are arrays that are indexed by the display window they control or belong to. This is a simple way of accessing the data for each window.

The first global array, "Pict", is an array of picture handles. In this program, only the text display window uses a picture, but we have allocated a picture handle for both. You can modify the program so that the graphics display window also uses a picture. However, this slows the graphics display scroll. We explain later some interesting techniques that allow us to draw faster than "pictures" can be drawn.

The next global array, "PicBnds", is an array of rectangles that frame the pictures. This picture "frame" is used when a picture is defined and

when it is displayed. Use the same frame in both instances if the picture is to be drawn undistorted.

The last global arrays are control handles. The first control array, "CtlButton", holds handles for the two control buttons that appear in our control window.

The arrays "hsbar" and "vsbar" hold handles to control the horizontal and vertical scroll bars on each of the two display windows.

Procedures

This program has several procedures, including those to initialize the system, to draw pictures that are displayed, to set up windows, and to handle events. Let's examine these procedures.

Setup

The first procedure, "SetUpSys", initializes QuickDraw, the Font Manager, the Window Manager, the Event Manager, and the cursor.

This procedure also allocates space for the region handle "URgn" by assigning it a value from the function "NewRgn". We do this once in the entire program, rather than each time the handle is used as Apple tends to do in the example programs that it provides with its development systems. This reduces the size of our program by a small amount.

Next, the "SetUpSys" procedure sets the delimiting values for the picture bounds, size bounds, and drag bounds rectangles. The picture bounds are arbitrarily set up for an entire screen size. The size and drag bounds are set up as in the previous program.

Lastly, this routine sets the "done" variable equal to FALSE. As in previous programs, "done" controls the main loop.

Drawing Stars

The next procedure, "Stars", draws an array of stars for the graphics window. Each star is drawn as a *polygon*, a QuickDraw data structure introduced earlier in the chapter.

The "Stars" procedure expects a single parameter that is a region handle to the region which delimits the portion of the window to be drawn. We draw only the stars that touch this region. This feature greatly increases the speed at which the display is drawn. Speed is important for fine scrolling.

Our procedure has a CONST section in which two constants, maxI and maxJ, are defined and both set to a value of 10. They determine the number of rows and columns in the array of stars.

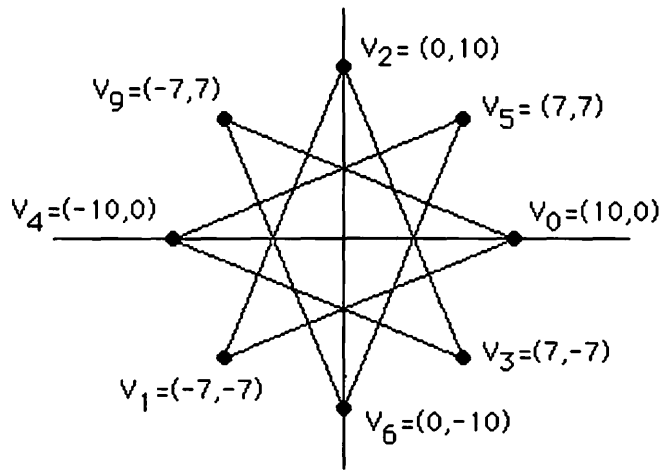
Our "Stars" procedure has several local variables declared in its VAR section. The first two are integers "I" and "J", which index through the star array. The last local variable, "Star", is a "polyHandle" that provides access to our "Star" polygon (see the earlier discussion on polygons).

The procedure begins by setting up the "Star" polygon. The first step is to use "OpenPoly" to "open" the polygon. This function returns a handle to the new polygon and begins saving all line drawing commands into this polygon. It also "hides" the pen so that no drawing shows on the screen while the polygon is open.

In our program, we call a series of QuickDraw's relative "Line" commands to form the polygon. The first "line" command places the two end points of the line in the polygon. The first point is the original position of the polygon when opened. The second point is the current position after the "line" command. Each subsequent line drawing command places the new position in the polygon (see Figure 7-6). The polygon is closed with the "ClosePoly" command.

Once the polygon is defined, we use the "OffsetPoly" routine to move it around, drawing at all the positions of the array. The first "OffsetPoly" moves it to the upper left corner of the array, where the first star is drawn. Then, a double FOR loop indexed by I and J runs through all rows and

Figure 7-6. The Star Polygon



columns of the array. "OffsetPoly" provides a relative motion of the polygon, making the basic structure of this double loop like that of the "PenSize" loop in the example program of Chapter 6.

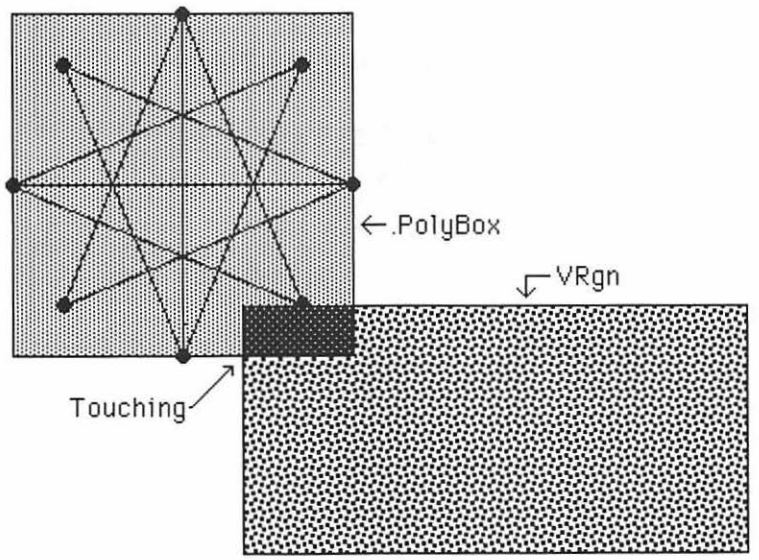
At each position we draw the polygon only if the polygon's ".polyBBox" touches the "VRgn" region that was passed to the routine (see Figure 7-7). This provides a quick way to ensure that we draw only those polygons needing to be updated.

A polygon's bounding box (given by its ".polyBBox") defines what is called the *extent* of the polygon (see Figure 7-8). More generally, the extent of any figure is a rectangle that specifies the limits of the figure (minimum and maximum horizontal and vertical coordinates). Since the extent is a rectangle, it is easier to work with than the original figure, providing a quicker test to determine if the figure falls within a certain area on the screen.

Since the extent is larger than the figure, it is possible for the extent to intersect an area even if the figure does not. Think of extent checking more as a way of eliminating parts of a picture that should not be drawn than as a way to find only those parts that need to be drawn.

Extent checking really speeds up the display in our program. There is a total of 100 stars, each containing eight line segments, thus making

Figure 7-7. "PolyBox" and "VRgn"



800 line segments. Drawing so many lines takes a significant fraction of a second. If we redraw the entire display for each step of a fine scroll, the scrolling appears sluggish. However, if we draw only the five to twelve stars that need updating each time, the drawing is eight to twenty times faster, and fine scrolling proceeds at an acceptable pace.

In our program, the "star" polygon is drawn with the "FramePoly" command. Other polygon drawing commands, such as "PaintPoly", "ErasePoly", "InvertPoly", and "FillPoly", produce results similar to the corresponding commands for other shapes, such as rectangles, ovals, and regions.

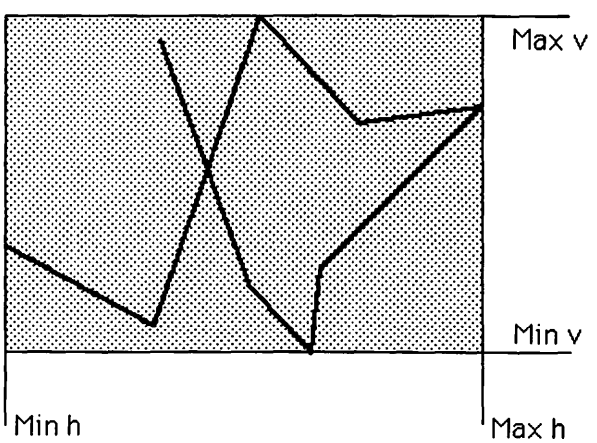
Drawing Text with Pictures

The procedure "Explain" draws the text displayed in the text window. The text is not located in the Pascal program itself but in a "string list" resource attached to the program.

In this procedure, we directly access the internal structure of the "string list" resource. In Chapter 10, we learn how the built-in routine "GetIndStr" makes this easier. Our procedure, which resembles the built-in routine, illustrates what a programmer would have to do if this routine were not available.

Our "Explain" procedure has several local variables. The first is "theText", a handle that accesses the "string list" resource. The second is "IPtr", a pointer to integers. The third is "S", a pointer to strings. The

Figure 7-8. The Extent of a Polygon



"IPtr" and "S" pointers access information in the "string list" resource. The last local variable is an integer used as a loop index.

The procedure begins with "GetResource" to access resource number 256 of type "STR#", where we store a multiline explanation of the program. The resource type "STR#" is a list of strings. The format of this resource in the resource file consists of the resource identification line, which in our case is simply:

```
,256
```

followed by as many lines of text as desired. This format allows us to freely compose whole blocks of text in the resource file. A blank line (a line of length zero or only spaces) terminates the text.

What the resource definition looks for in this string list reads like an introductory explanation to our example program. This is the same text that we see in our text window:

```
,256
     This program illustrates how to manage several windows. A
control window has button controls that make two other windows
appear.  One of these windows displays an array of stars, the
other displays this explanation.  Both windows can be sized and
scrolled.  The control window has a goAway box that ends the
program.  The other two windows each contain a goAway box that
makes its particular window disappear.
```

The "GetResource" function returns a handle to the resource. We store this handle in the variable "theText". Then, we call the "HLock" procedure to "lock" the block of memory containing the text data so that it doesn't move as we directly access it with our own pointers. Otherwise, the text might get garbled.

Once the text is locked, we set up our pointers to get the data. The internal format of the resource consists of an integer that specifies how many strings are in the list, followed directly by the strings. Each string consists of a byte that gives its current size, followed by the bytes containing the ASCII codes of the strings. In our program, we first point "IPtr" to the beginning of the resource so that it points to the integer specifying the number of strings. Then, we point "S" to the first string, which is two bytes later.

A FOR loop indexed by i runs through all strings in the resource. Within the loop, we use the "MoveTo" procedure to move to the beginning of the line on the screen and the "DrawString" procedure to draw the text. We then adjust the string pointer "S" to point to the next string in

the list. After the FOR loop, we call "HUnLock", which allows the text resource to be dynamically moved in memory by the memory manager.

Setting Windows

The procedure "SetUpWindows" initializes all three windows. It begins with the "GetNewWindow" routine to get the control window, then calls "GetNewControl" twice to get the two button controls.

Here is the resource definition for the control window:

```
,257
Control Window
40 10 80 198
Visible goAway
16
0
```

The first line of the control window resource definition is its identification number; in this case, 257. The second line is its title, "Control Window". The third line delimits its port rectangle in global coordinates. The fourth line specifies that it is visible and has a goAway box. The fifth line specifies an identification number for its window definition procedure. In this case, a value of 16 indicates a rounded-corner window (see Figure 7-9). (In Chapter 6, we listed other possibilities.) The sixth line specifies the reference number. For the control window, the reference number is set to zero.

Here are the resource definitions for the button controls:

```
Type CNTL
  ,257
  graphics
  10 20 30 100
  Visible
  0
  0
  0 0 0
```

Figure 7-9. Rounded-Corner Window

```
,258
text
10 120 30 168
Visible
0
0
0 0 0
```

The first line of the resource for the graphics control button is its resource identification number; in this case, 257. The second line is the title, "graphics". The third line is the limits in local coordinates. The fourth line specifies that the control is visible. The fifth line is the control definition procedure; in this case, zero, indicating a standard button. The sixth line specifies its reference number, which is zero. We change this field in our Pascal program. The seventh line gives the initial values for the control's current value, minimum value, and maximum value. All three are zero for this button.

The text control button has a similar resource. The title and location are different, but the other lines are the same.

We store handles to these resources in "CtlWindow", "CtlButton[1]", and "CtlButton[2]", respectively. Instead of a regular window, we could use a dialog box for our control window, but we want to illustrate that dialog boxes are simply windows with a few extra "bells and whistles". Chapter 8 is devoted to dialogs.

Next, we set up the window and scrolling controls for the graphics window. Again, we use "GetNewWindow" and "GetNewControl" to get the initial parameters from the resource file. Here is the resource definition for this window:

```
Type WIND
  ,258
  Graphics Window
  100 100 200 400
  Invisible goAway
  0
  1
```

Notice that it is a standard document window (see Figure 7-10).

Here are the resource definitions for the scroll controls for the graphics window:

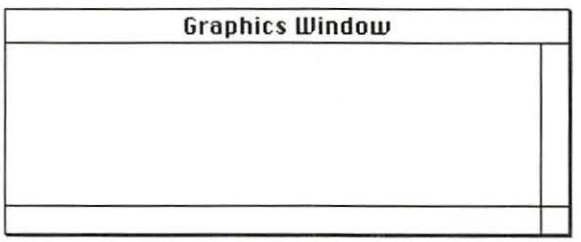
```
Type CNTL
  ,259
  horizontal scroll bar
```

```
85 -1 101 286
Invisible
16
1
0 0 450

,260
vertical scroll bar
-1 285 86 301
Invisible
16
2
0 0 450
```

The graphics window uses the generic "theWindow" to temporarily store its window pointer. We immediately place the address of this pointer in the reference field of the first control button and in the "owner" fields of its scroll bars. We can then reuse the pointer "theWindow" for other purposes. This conserves the number of variables in the program, thereby shortening and simplifying the program. We can always recover a pointer to the graphics window by accessing any of these three fields.

Finally, we set up the text window. We grab the window and scrolling controls as before and also create the *picture* for the text window. We call "SetPort" to make sure that the current grafPort is the text window's grafPort. Then, we call "ClipRect" to set the clipping region to the picture bounds. This is neccessary because the limits of the clipping region are stored in the picture definition data. The default value is a maximum clipping area, stretching from –32767 to 32767 in both horizontal and vertical directions. If you use the default setting and then try to move or expand the picture by drawing it to another rectangle, you will run into overflow problems. To avoid this, always clip before you define the picture.

Figure 7-10. Standard Document Window



Now we call "OpenPicture" to start saving the picture into the picture handle "Pict[2]". We call the "Explain" subroutine to draw the text into the "opened" picture, then call "ClosePicture" to end the picture-making process. Here are the resource definitions for the text window and its controls:

```
Type WIND
  ,259
  Text Window
  200 60 300 360
  Invisible goAway
  0
  2


Type CNTL
  ,261
  horizontal scroll bar
  85 -1 101 286
  Invisible
  16
  1
  0 0 400

  ,262
  vertical scroll bar
  -1 285 86 301
  Invisible
  16
  2
  0 0 120
```

Look through the resources for these three windows. Notice that the control window is defined to have reference number 0, the graphics window to have reference number 1, and the text window to have reference number 2. These reference numbers play an important role in our program, allowing us to quickly determine which window we are dealing with.

Updating Pictures

The next procedure, "UpdatePic", draws and redraws the contents of the display windows as needed for scrolling and window updating.

The procedure expects two parameters: an integer, "i", that identifies which window needs updating, and a region handle, "URgn", that specifies which region needs updating.

The "UpdatePic" procedure has a single local variable, "S", of type "Point". This variable temporarily holds the displacement for scrolling.

The procedure begins by checking the specified window. If it is the control window, it exits immediately. This statement, though not absolutely necessary, is good programming practice, since certain statements in the procedure demand that integer "i" take on values in the range 1..2. This precaution helps if we change the program in such a way that "i" takes on other values upon entry to the routine.

We next call "SetPt" to set the horizontal and vertical components of "S" equal to the displacements stored in the control values for the horizontal and vertical scroll bars that belong to the indicated display window.

We then call "SetOrigin" to translate the local coordinate system by "S" followed by "OffsetRgn" to translate the "URgn" by the same amount. This takes care of the accumulated effects of scrolling and keeps the specified region in the same area of the window.

Now we use "SetClip" to set the clipping region equal to the specified region. This ensures that only those parts needing to be drawn are drawn. We then erase the region. Otherwise, when we scroll to the edges of our drawings, the new parts do not completely fill the window; thus, they do not completely cover the outdated parts.

A CASE statement now determines which of the two display windows is being updated. For the graphics window (assigned a reference value of one), we call the "Stars" procedure. For the text window, we draw the picture stored under the picture handle Pict[2].

Notice that we call the Memory Manager's "HLock" routine before, and its "HUnLock" routine after, the "DrawPicture" command. Always surround the "DrawPicture" command in this way to ensure that the picture pointers do not tangle during the picture-drawing process.

The "HLock" and "HUnLock" routines expect a parameter of type "Handle", but we want to pass a picture handle (type "PicHandle"). The Pascal compiler objects if we do this directly. One solution: use the ORD function to convert the picture handle to a long integer (its address), then use the POINTER function to convert this address into a neutral type of pointer that can be passed as a "Handle" or any other type of pointer.

In our program, we solve the typing problem by a feature of the Pascal compiler called "type coercion", described in Chapter 3. This method uses the identifier of the desired type as a function to operate on an expression of the original type, returning a value of the desired type. For example, "Handle(Pict[2])" is an expression of type "Handle" that has been "coerced" from "Pict[2]", which is an expression of type "PicHandle".

Once the picture is drawn, we use "SetOrigin" to place the origin back to (0,0). If we leave the origin where it is, the scrolling controls do

not work properly. Finally, we set the clipping region equal to the entire port rectangle so that the scroll bars are properly highlighted when selected.

Scrolling

"ScrAction" is the action procedure for scrolling. The Control Manager calls this procedure as it tracks a control. It updates the scrolling control values and scrolls the display as part of the tracking process.

Our routine is a generalization of the scroll action procedure from Chapter 6. It has the same parameters, namely a control handle, "theCtl", and an integer, "partcode", that specifies which part of the control is selected.

The procedure has several local variables. The variables "pageSize" and "delta" are integers that determine the amount of scrolling. The variable "i" is an integer that specifies which window is being scrolled. The variables "S" and "dS" are of type "Point" and determine the displacement and change of displacement for scrolling. The variable "viewBnds" is a rectangle that clips the drawing area of the window.

The routine begins by loading the reference value of "theWindow" into the variable "i". The window pointer "theWindow" is the currently selected window. After this statement, we can use "i" to drive CASE statements and IF statements to quickly determine which window is selected and therefore which action is required.

If "theWindow" is the control window (i = 0), we call "Exit" to leave the routine because there is no scrolling for the control window. Again, as in the "UpdatePic" routine, we protect against this case, even though it is not supposed to happen.

The next section repeats the action routine of the program in Chapter 6. It determines whether the horizontal or the vertical scroll bar has been selected and sets the page size accordingly. If neither is selected, we call "Exit". When we find out which part of the scroll bar is selected, we set "delta" to the indicated change in horizontal or vertical displacement values for scrolling. If no valid part is being tracked, we call "Exit" and leave the routine without further action.

Next, we update the control values for the scroll bars and determine the actual change in displacement values for scrolling. The *actual* change differs from the *indicated* change when scrolling displacement reaches its minimum or maximum limits. In these cases, the button control should be tracked, but the corresponding control value should not exceed the delimiting value.

We use the built-in facilities of the Control Manager to properly handle the control limits. First, we read the values from the scroll controls

into the local variable "S" to temporarily save the original values. Then we update the controls by adding delta to the selected control value. Now we compute the difference between the original value as stored in "S" and the new control values. The result is placed in dS.

We can now call QuickDraw's "ScrollRect" to perform any fine scrolling by shuffling bits on the screen. We then call our own "UpdatePic" to redraw those parts of the picture that have come into view as a result of the scroll. This completes the scroll.

Control Management

The procedure "WindowControl" performs the general management of controls. It expects one parameter, a point we call "thePt". This is the position of the mouse when the mouse button is pressed.

The procedure has one local variable: a control handle, "theCtl", which is used in the control selection process.

The procedure begins by checking whether the mouse is pressed in the "front" window. This is the currently active window, in front of the other windows. If the mouse is in the active window, we track the controls. Otherwise, we make the selected window into the currently active window.

If we decide to track the controls, we use "SetPort" to set the grafPort to the grafPort of the active window. We use "GlobaltoLocal" to transform the coordinates of the mouse point to the local coordinates of "theWindow", then use the CASE statement to determine which part of which control is selected (if any).

If the "inButton" is selected, we are in the control window because other windows have no regular buttons. Two control buttons are in the control window: one for the graphics window, one for the text window. In this case, we call "TrackControl" to track the selected control. This highlights the button when the mouse is in the button region. For the control buttons, we don't want any special programmer action routine, so we set the third parameter of "TrackControl" equal to NIL.

If the tracking function returns a nonzero result, the mouse has not left the selected control and we can safely perform the indicated action. In this case, we want to "unhighlight" the control button and make the corresponding display window visible, if it is not already so.

If the "inUpButton", "inDownButton", "inPageUp", or "inPage Down" control is chosen, we are in one of the scroll bars. In this case, we need to track the control with our special action routine, which we pass as the third parameter to the "TrackControl" routine. When this tracking routine returns, we don't need to perform any action. All special action is performed within the tracking by our "ScrAction" routine.

If the "inThumb" control part is selected, we track the control with no action routine, but we update the picture when we finish tracking. Before calling "UpdatePic" we set the update region, as specified by "URgn", equal to the entire viewing area so that the entire picture is redrawn.

This covers all cases when the selected window is the front window. If the selected window is not the front window, we execute the "ELSE" part. We call "SelectWindow" to highlight the selected window and bring it to the front. We call "DrawGrowIcon" to redraw its grow icon, and call "DrawControls" to redraw its controls.

Sizing Windows

The procedure "WindowGrow" lets the user resize a selected display window. It expects one parameter: an integer, "i", that specifies which of the three windows is selected. It has two local variables: WSize, a long integer to hold the horizontal and vertical components of the new window size; and a point, "S", a more convenient way of holding this same information.

The routine begins by calling "GrowWindow" to track the grow icon and return the new size in "WSize". If the window's size has not changed, we call "Exit" and leave the routine. Otherwise, we proceed. We call "SetPt" to convert the size into the form of a point in "S". Then, we call "SizeWindow" to resize the window.

Once the window is resized, we need to redraw its scroll controls. First we call "SetPort" to make sure that we draw to the right grafPort. We set the clipping limits to the new window size, then resize and move the scrolling controls. This resembles the window sizing routine in Chapter 7.

The GoAway Box

The procedure "WindowGoAway" handles the goAway box. It expects a single integer parameter, "i", which specifies one of the three windows.

The procedure calls "TrackGoAway" to track the goAway box. If this Boolean function returns a TRUE, then the mouse stayed in the box when the button was released, and we take appropriate action. For the control window, we set "done" equal to TRUE. For a display window, we call "HideWindow" to make it disappear and "HiliteControl" to "undim" the corresponding control button. According to Apple, a button should remain "undimmed" as long as it can do something and "dimmed" if it cannot.

In this case, the button is "undimmed" because it is able to bring the display window back into view.

Updating Windows

The procedure "WindowUpdate" provides the overall management of updating windows. It properly updates the grow box, the window controls, and the picture in the window when an update or activate event occurs.

The procedure expects one parameter: an integer, "i", that specifies which window is being updated. It has one local parameter, "growarea", which delineates the grow box. This updates the grow box each time the window is updated.

The procedure begins by calling "SetPort" in preparation for drawing to the window. If the selected window is not the control window, we perform a series of tasks. First, we set the position of the "growarea" relative to the port rectangle. A WITH statement helps to shorten the formulas. We add the "growarea" to the update region by calling "InvalRect". If the selected display window is the front window, we call "ShowControl" to ensure that its scrolling controls are visible. Otherwise, we call "HideControl" to ensure that they are invisible.

We now perform a series of tasks for all windows. We call "BeginUpdate" to start the updating process. This sets the visRegion equal to the update region and makes the update region empty. Again, we invert the port rectangle and "beep" in order to see what is being updated. Of course, these two steps are removed in an actual application.

Next, we erase the window (only the updated part will erase), redraw the controls and grow box, and update the picture. Finally, we call "EndUpdate" to restore the VisRegion.

The Main Program

The main program begins by calling "SetUpSys" to initialize the system and "SetUpWindows" to initialize the windows and their controls.

Most of the main program consists of a REPEAT loop that sorts the events and calls the approriate procedures to handle them. It is similar to the main loop of the example program in Chapter 6.

The REPEAT loop begins by calling the "GetNextEvent" function in an IF statement. If this function returns true, we must handle the event. We use a CASE statement to examine the ".what" field.

The first case of ".what" is "mouseDown". We check ".where" the mouse is. If it is "inDesk", we simply select the control window. If it is "inContent", we call our "WindowControl" procedure to check for the

selection of controls. If it is "inDrag", we call "DragWindow" to drag the window to a new location. If it is "inGrow", we call our "WindowGrow" procedure to resize the window and its contents. If it is "inGoAway", we call our "WindowGoAway" procedure to handle the goAway boxes.

The second cases of ".what" are "UpdateEvt" and "ActivateEvt". Both are handled by setting "theWindow" equal to the message field for these window updating events and calling our "WindowUpdate" procedure to handle the updating, as described previously.

The REPEAT loop continues looping through these events until the Boolean "done" becomes true (whenever the goAway box of the control window is selected).

Summary

In this chapter, we have extended our knowledge of window management to handle multiple windows. We have seen how to manage multiple graphics and text windows as well as multiple button and scroll controls. We have seen how *pictures* and *polygons* can help us draw pictures faster and easier. We have also seen how to make windows and their controls appear and disappear at our command.

The following ROM routines were covered in this chapter:

QD-OpenPoly

QD-Line

QD-ClosePoly

QD-OffsetPoly

QD-RectInRgn

QD-FramePoly

MM-HLock

MM-HUnLock

CM-SetCRefCon

QD-OpenPicture

QD-ClosePicture

QD-SetOrigin

QD-OffsetRgn

QD-DrawPicture

WM-GetWRefCon

QD-ScrollRect

WM-FrontWindow
CM-HiliteControl
WM-ShowWindow
WM-SelectWindow
WM-HideWindow
WM-InvalRect
CM-ShowControl
CM-HideControl

8

Dialogs and Alerts

This chapter covers the following new concepts:

- **The Dialog Manager**
- **Dialogs and Alerts**
- **Modal and Modeless Dialogs**
- **Dialog Record Structure**
- **Dialog Item Lists**
- **Setting Up Dialogs**
- **Tracking Dialogs**
- **Dialog Text and Control Items**
- **Types of Alerts**

In this chapter, we explore dialogs and alerts. From the user's point of view, a dialog box is a convenient way to enter a variety of different types of input, including immediate actions, setting of Boolean variables, selection from a few alternatives, and fully edited text. Programmers find dialogs convenient because they require minimal effort to program fully developed standard control structures for input of vital program data.

To prove how convenient dialogs are to both the user and the program, we present an extremely short example program that manages control structures that can be successfully operated by a three-year-old child.

Dialogs and alerts are managed by the Dialog Manager. This manager operates at a higher level than many managers in the Macintosh. In fact,

certain of its alert routines are like mini-applications. These routines first draw a window with controls, then loop around, repeatedly calling the Event Manager's "GetNextEvent" routine and the Window Manager's "FindWindow" routine to track the mouse as it selects its control items.

The Dialog Manager has a number of levels, just like an applications program. At the lowest levels, it calls the Window Manager, the Control Manager, the Event Manager, Text Edit, the Desk Manager, and the Resource Manager. At higher levels, it mainly calls its own lower-level routines.

Dialogs and Alerts

Dialogs and alerts are special forms of interaction between the program and the user. They can be thought of as *processes* that have their own data structures and execute for awhile, like programs, in the machine.

During a dialog or alert, information from the Macintosh to the user appears in a *dialog box* or *alert box*. The user inputs information in the usual way through the mouse and the keyboard.

Dialogs and alerts are managed by the Macintosh's Dialog Manager, which, like the other managers, consists of routines and data structures in the Macintosh's memory.

Dialog and alert boxes are implemented as windows (see Chapters 6 and 7) and usually have controls associated with them. In addition to controls, a dialog or alert can have other entities, such as text strings, icons, and pictures. Associated with each dialog and alert is a list of all entities under its control. This *dialog list* is normally specified in the program's resource file, and the entities it contains are called *dialog items*.

Much of the control tracking can be automatically handled by the *Dialog Manager*. However, a programmer can substitute a custom routine, called a filter, that handles these controls and other entities in the dialog or alert. This feature is not discussed here because the standard built-in routines are adequate for most purposes. However, an example of such a routine is contained in Appendix C.

Comparison of Dialogs and Alerts

The difference between dialogs and alerts is in the amount of information they return to the program that calls them and when that information is returned.

Alerts are designed to return only a single selection variable. Their main purpose is to communicate a special message to the user, usually an error or exception message (see Figure 8-1). Normally, the only infor-

mation returned by the user after an alert is the item number of the item selected. Often, this is just the "OK" button.

Dialogs can return several kinds of information, including Boolean variables, selections from a small number of possibilities, and fully edited text. There are two forms of dialogs: *modal dialogs* and *modeless dialogs* (see Figure 8-2). A *modal dialog* takes over all communication between the user and the Macintosh while it is active. In contrast, a *modeless dialog* allows other information to be exchanged while the dialog is active. In both cases, the applications program loops, repeatedly calling for information from the dialog and potentially using it while the dialog is still active. This contrasts to an alert, which disappears as soon as it returns information to the program.

In this chapter, we explore *modal dialogs* and *alerts* with our example program. We see how just a few calls allow a programmer to set up a dialog or an alert, leaving most of the work to the Dialog Manager's routines, which deliver the results. Thus, tasks that we had to program ourselves in previous chapters are now done automatically in the example program here.

Figure 8-1. An Alert Box

The Example Program

The example program in Chapter 8 presents a dialog box with a multitude of buttons, boxes, and labels (see Figure 8-3). The program also displays two alert boxes with special system icons and our own messages.

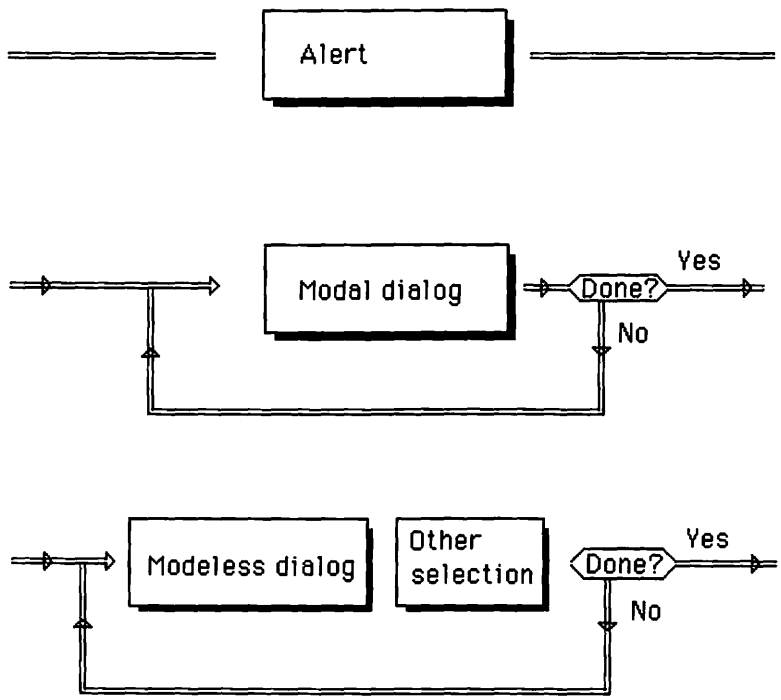
These controls are not connected to a true application; however, they can easily supply several types of data to an application.

The programs open by drawing a large, empty dialog box on the screen. The dialog box is outlined with a double border consisting of a thick frame line surrounded by a thinner frame line.

In the lower left corner of the dialog box are two buttons: the "OK button" and the "cancel" button. If you press either button, the dialog ends. Pressing the OK button retains any selection made during the dialog. Pressing the cancel button exits without saving any selection.

At the top of the screen is a message: "Modal Dialog Demonstration" and "Type "quit" to exit.". Below this message is a box to enter text. If you strike keys on the keyboard, the corresponding characters appear in

Figure 8-2. Alerts, Modal Dialogs, and Modeless Dialogs

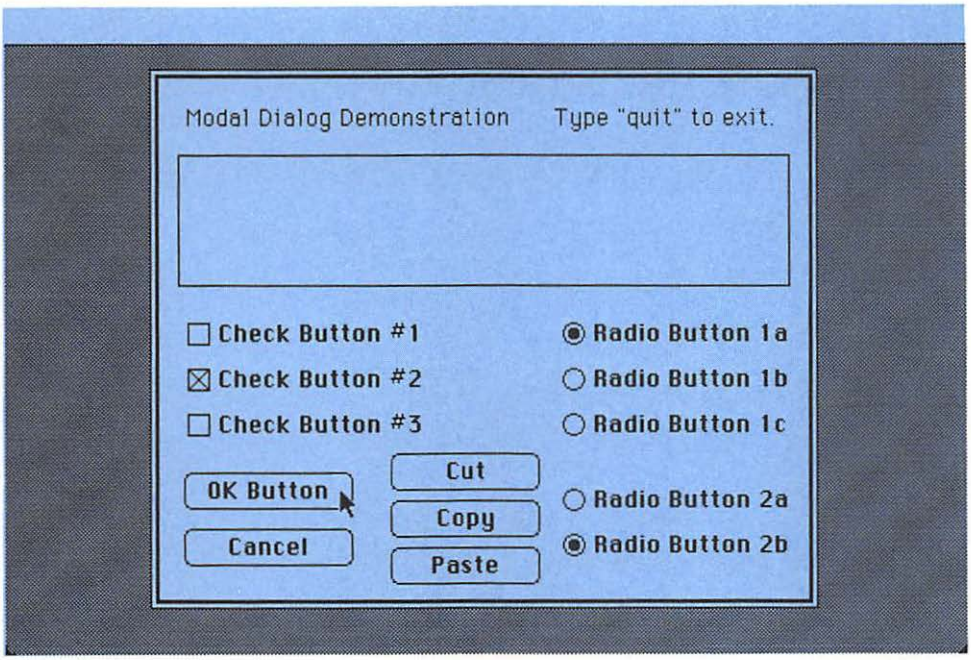


this box as a text string. You may edit this text in the normal way, hitting backspace to erase the last character, using the mouse to move the insertion point, or using the mouse to select a portion of text to be erased by hitting the backspace. In addition, "cut", "copy", and "paste" buttons in the bottom center of the dialog box allow you to perform the indicated action on this text.

On the left side of the dialog box, about halfway down, are three check boxes. You can "toggle" the value of each check box; that is, each check box is either unchecked (blank) or checked (filled with an "x"). You switch between these two states by clicking the mouse in the box; one click "checks" the box, another click unchecks it, a third checks it again, and so on. That is, each check box acts like a *Boolean* variable that can assume only two values.

On the lower right side of the dialog box are two groups of "radio buttons". These buttons behave like the buttons on a car radio; that is, pressing any button selects that button and "deselects" all others in the same group. This gives a way to select one option out of several: each group of radio buttons acts like an *integer* variable that can assume a small number of values. The radio buttons in the first group are labeled

Figure 8-3. The Dialog Box Filled with Items



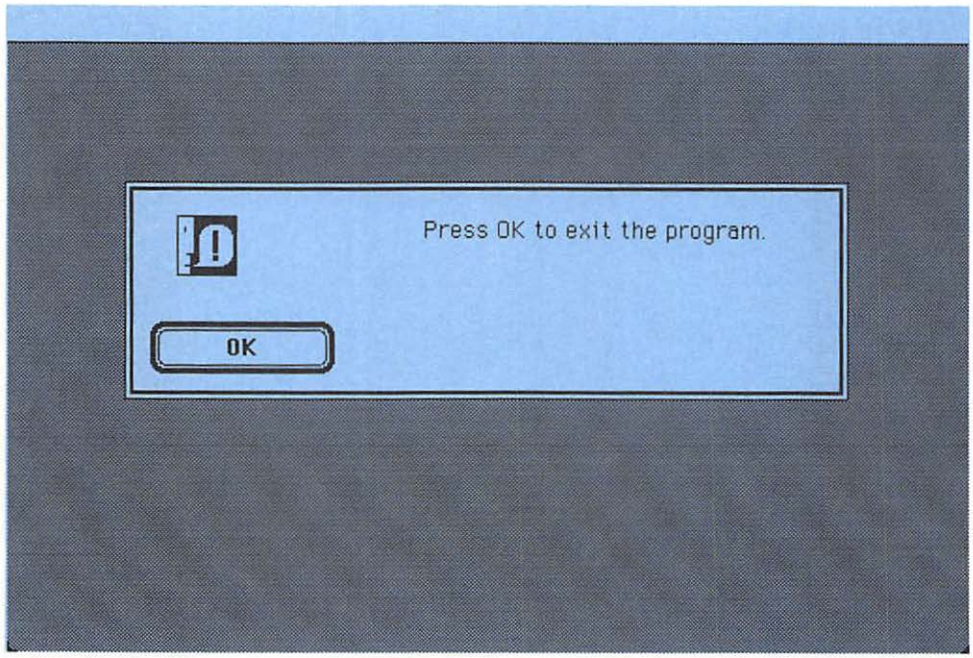
"radio button 1a", "radio button 1b", and "radio button 1c"; the second group of radio buttons are labeled "radio button 2a" and "radio button 2b".

You end the dialog by selecting the OK button or cancel button or by hitting the Enter or Return keys on the keyboard. The dialog box then disappears. If you type the word "quit" in the first four spaces of the text box, an alert box appears with a "stop" icon, stating that you can press its OK button to end the program (see Figure 8-4). You are committed to end the program once you get to this point. You have only two choices: select the OK button or hit the reset button. In either case, the program terminates. This mimics the way many programs bomb out on the Macintosh. Perhaps, in this case, the OK button ought to be called the "Got Ya" button. We see later how to alleviate this situation.

If the "quit" condition is not met, then an alert box appears with a "note" alert. The message states that the values entered from the dialog can now be used by an application (see Figure 8-5). If you select the OK button in this alert box, you return to the original dialog.

If we don't exit right away, then we can change the settings and the text in various ways (see Figure 8-6).

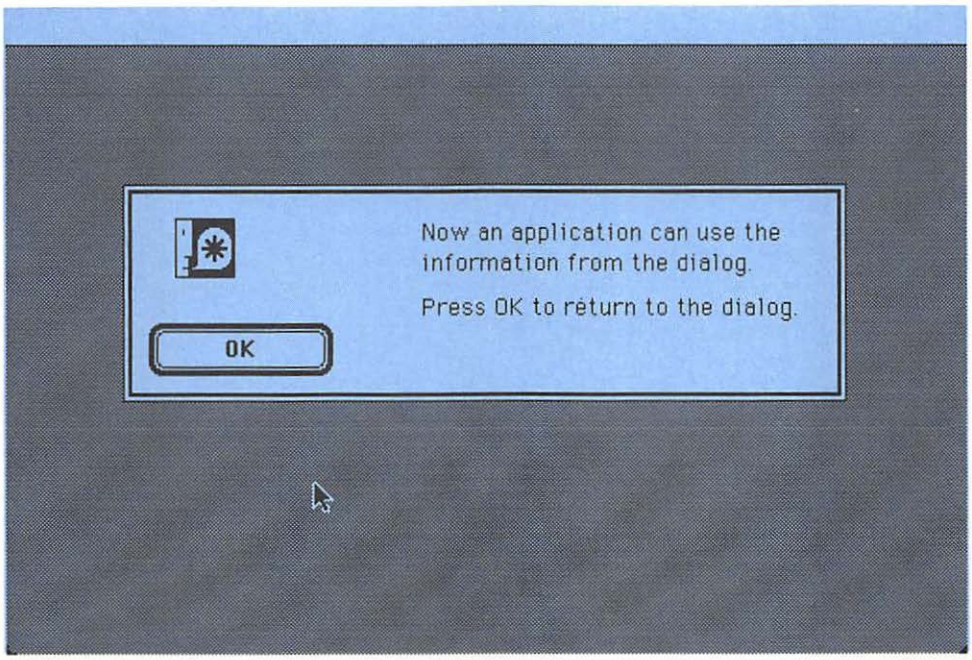
Figure 8-4. The Stop Alert



Here is the program:

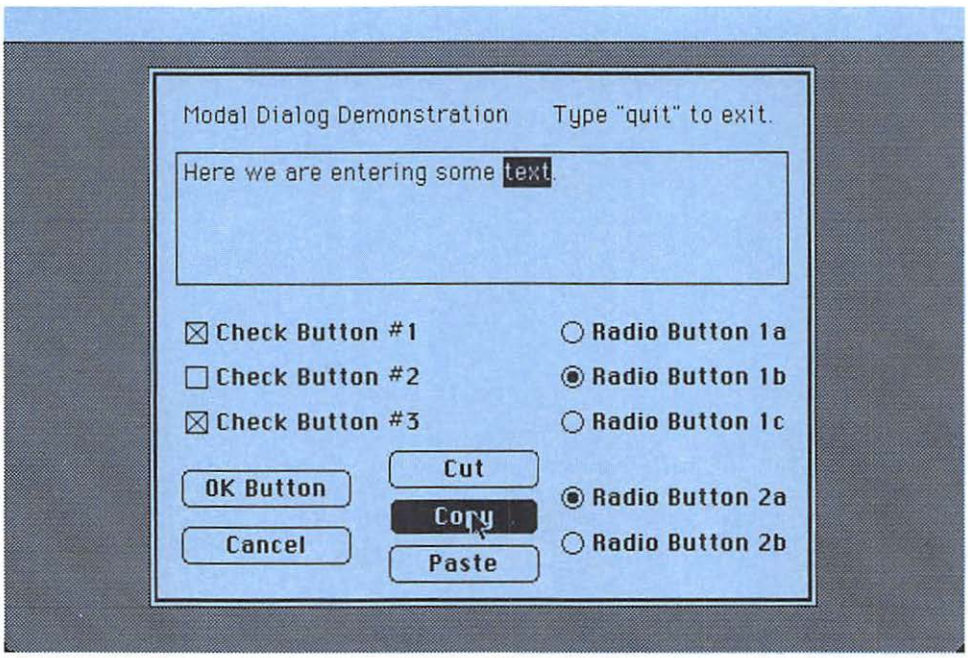
```
PROGRAM DialogDemo;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf;

   CONST
      {Dialog items}
      OKBtn     = 1;
      cancelBtn = 2;
      statTxt   = 3;
      edTxt     = 4;
      cutBtn    = 5;
      copyBtn   = 6;
      pasteBtn  = 7;
      chkBtn1   = 8;
      chkBtn2   = 9;
```

Figure 8-5. The Note Alert



```
    chkBtn3  = 10;
    radBtn1a = 11;
    radBtn1b = 12;
    radBtn1c = 13;
    radBtn2a = 14;
    radBtn2b = 15;
    numCButtons = 3;
    numRGroups  = 2;

VAR
    done: BOOLEAN;
    theDialog: DialogPtr;
    theType, ItemHit, theItem: INTEGER;
    ItemHdl: Handle;
    ItemBox: Rect;
    theText: STR255;
    CArray: ARRAY [1..numCButtons] OF BOOLEAN;
    RGroup: ARRAY [1..numRGroups] OF
              RECORD
                 min, max, default: INTEGER
              END;
```

Figure 8-6. Dialog Box After Some Changes



```
PROCEDURE SetUpSys;
   BEGIN
      InitGraf(@thePort);
      InitFonts;
      InitWindows;
      TEInit;
      InitDialogs(NIL);
      SetEventMask(everyEvent);
      SetDAFont(1);
   END;

PROCEDURE SetDefaults;
   BEGIN
      theText := '';
      CArray[1] := FALSE;
      CArray[2] := TRUE;
      CArray[3] := FALSE;
      RGroup[1].min     := radBtn1a;
      RGroup[1].max     := radBtn1c;
      RGroup[1].default := radBtn1a;
      RGroup[2].min     := radBtn2a;
      RGroup[2].max     := radBtn2b;
      RGroup[2].default := radBtn2b;
   END;

FUNCTION CtlHdl(theItem: INTEGER): ControlHandle;
   BEGIN
      GetDItem(theDialog,theItem,theType,ItemHdl,ItemBox);
      CtlHdl := ControlHandle(ItemHdl);
   END;

PROCEDURE SetUpDialog;
   VAR
      I, J : INTEGER;
   BEGIN
      theDialog := GetNewDialog(1000,NIL,POINTER(-1));
      For I := ChkBtn1 TO ChkBtn3 DO
         SetCtlValue(CtlHdl(I),ORD(CArray[I-chkBtn1+1]));
      For I := 1 to numRGroups DO
         For J := RGroup[I].min TO RGroup[I].max DO BEGIN
            SetCRefCon(CtlHdl(J),I);
            SetCtlValue(CtlHdl(J),ORD(J = RGroup[I].default));
            END;
      SetIText(Handle(CtlHdl(EdTxt)),theText);
      SelIText(theDialog,EdTxt,length(theText),length(theText));
   END;
```

```
PROCEDURE SetStdBtn(theItem: INTEGER);
   BEGIN
      CASE theItem OF
         cutBtn:   DlgCut(theDialog);
         copyBtn:  DlgCopy(theDialog);
         pasteBtn: DlgPaste(theDialog);
         END;
   END;

PROCEDURE SetChkBox(theItem: INTEGER);
   BEGIN
      SetCtlValue(CtlHdl(theItem),1-GetCtlValue(CtlHdl(theItem)));
   END;

PROCEDURE SetRadBtn(theItem: INTEGER);
   VAR
      I, J : INTEGER;
   BEGIN
      I := GetCRefCon(CtlHdl(theItem));
      FOR J := RGroup[I].min TO RGroup[I].max DO
         SetCtlValue(CtlHdl(J),ORD(J = theItem));
   END;

PROCEDURE UpdateDefaults;
   VAR
      I, J : INTEGER;
   BEGIN
      FOR I := 1 TO numCButtons DO
         CArray[I] := GetCtlValue(CtlHdl(chkBtn1+I-1)) <> 0;
      FOR I := 1 to numRGroups DO
         FOR J := RGroup[I].min TO RGroup[I].max DO
            IF GetCtlValue(CtlHdl(J)) = 1
               THEN RGroup[I].default := J;
      GetIText(Handle(ctlHdl(EdTxt)),theText);
   END;

PROCEDURE DoDialog;
   BEGIN
      FlushEvents(everyEvent,0);
      REPEAT
         ModalDialog(NIL,itemHit);
         CASE itemHit OF
            cutBtn..pasteBtn:   SetStdBtn(itemHit);
            chkBtn1..chkBtn3:   SetChkBox(itemHit);
            radBtn1a..radBtn2b: SetRadBtn(itemHit);
            END;
      UNTIL itemHit in [OKBtn,CancelBtn];
      IF itemHit = OKBtn THEN UpdateDefaults;
      done := (theText = 'quit');
   END;
```

```
BEGIN {main}
   SetUpSys;
   SetDefaults;
   InitCursor;
   REPEAT
      SetUpDialog;
      DoDialog;
      DisposDialog(theDialog);
      If done THEN theItem := StopAlert(1001,NIL)
             ELSE theItem := NoteAlert(1002,NIL);
   UNTIL done;
END.
```

Data Structures

The "DialogDemo" opens with the standard USES section.

Global Constants

In this program, the CONST section provides a convenient interface between the resource definition file and the program. It lists all items that belong to the dialog in the precise order that they appear in the resource definition file. In addition, two constants give the number of check buttons and the number of radio button groups. This list was extremely valuable during the development of this program. As we rearranged and added items, we found that the program still worked perfectly as long as we updated the constants for each change.

Let's look at these constants in detail.

The first two global constants, "OKbtn" and "cancelBtn", indicate the position in the resource list of the OK button and the cancel button.

The global constant "statTxt" gives the list position (position in the dialog list) of the title message. Then the "edTxt" global constant gives the list position of the editable text that appears in the text box.

The constants "cutBtn", "copyBtn", and "pasteBtn" describe the list positions of the cut, copy, and paste buttons used to edit our text.

The constants "chkBtn1", "chkBtn2", and "chkBtn3" give the list positions of the check buttons. The constants "radBtn1a", "radBtn1b", and "radBtn1c" give the list positions of the first group of radio buttons, and constants "radBtn2a" and "radBtn1c" give the list positions of the second set of radio buttons. All radio buttons must appear in order and together in the list for the program to work properly.

The constant "numCButtons" gives the number of check buttons, which is three. The constant "numRGroups" gives the number of radio button groups, which is two.

Global Variables

The VAR section declares several global variables.

Global variable "done" is a Boolean variable used for loop control, as in earlier programs.

Global variable "theDialog" is a dialog pointer that locates the dialog's data structure.

The next three global variables are integers. Integer "theType" holds a type code for the dialog items. Table 8-1 lists these type codes. Notice that this type code allows us to distinguish different kinds of buttons, text items, and other items. This feature is not used in our program.

Integer "ItemHit" holds the list position of a selected item. Both "theType" and "ItemHit" are returned from the Dialog Manager.

Integer "theItem" is used as a "dummy" variable when the alerts are called.

Global variable "ItemHdl" is a handle returned from the Dialog Manager to provide access to the data structures of dialog resource items.

Global variable "ItemBox" is a rectangle that is returned by the Dialog Manager and gives the "extent" or bounding box for a selected dialog item. This is called the item's display box.

Global variable "theText" is a string that holds the text from the text box when requested.

The remaining global variables are arrays that hold values from the check and radio buttons. "CArray" is an array of Boolean variables that holds values from the check buttons. "RArray" is an array of records that

Table 8-1. Item Type Codes

Type	*Code*
ctrlItem	4 (Add to following four)
Ctrl	0
chkCtrl	1
tradCtrl	2
tresCtrl	3
statText	8
editText	16
iconItem	32
picItem	0
itemDisable	128

holds "minimum", "maximum", and "default" position values for each group of radio buttons. As global variables, they retain values from the dialog even after the dialog has been disposed of.

Dialog and Alert Data Structures

From the point of view of data structures, a dialog or an alert is an extension of a window, which is an extension of a grafPort. That is, each dialog contains a window, and each window contains a grafPort. Like windows, dialogs and alerts are accessed through pointers, not handles, and therefore are stored in nonrelocatable blocks in the heap.

Also as with windows, there are two levels of pointers to the data. One grants access only to the grafPort section of the data. The other, a "peek" variable, grants access to the entire structure. These pointers and structures are defined through the following Pascal declarations:

```
DialogPtr =  WindowPtr;
DialogPeek = ^DialogRecord;
DialogRecord = RECORD
window:      WindowRecord;
items:       Handle;
textH:       TEHandle;
editField:   INTEGER;
editOpen:    INTEGER;
aDefItem:    INTEGER;
END;
```

Here, the type "DialogPtr" is defined as equal to the type "WindowPtr", which was previously defined as equal to the type "GrafPtr". Here also, the type "DialogPeek" is a pointer to a variable of type "DialogRecord", containing the entire record structure that houses the data.

The dialog record contains several fields (see Figure 8-7). The first field, ".window", is the dialog's window record structure. The second field, ".items", is a handle to the item list for the dialog.

The next three fields are used with editable text items and apply to the current editable text item (if any). This text item displays the insertion point.

The ".textH" field is a handle to the current editable text. The ".editField" is the item number of the current editable text in the dialog list. It contains one less than its position number. The ".editOpen" field is for internal purposes.

The ".aDefItem" field contains the item number of the button used to exit the dialog. This is called the default button and is usually the OK

button but sometimes the cancel button. For modal dialogs, this field has a value equal to one, indicating the first item in the list. For alerts, which are treated internally as modal dialogs, this field is one or two depending on staging information supplied by the alert's resource definition. We examine this later.

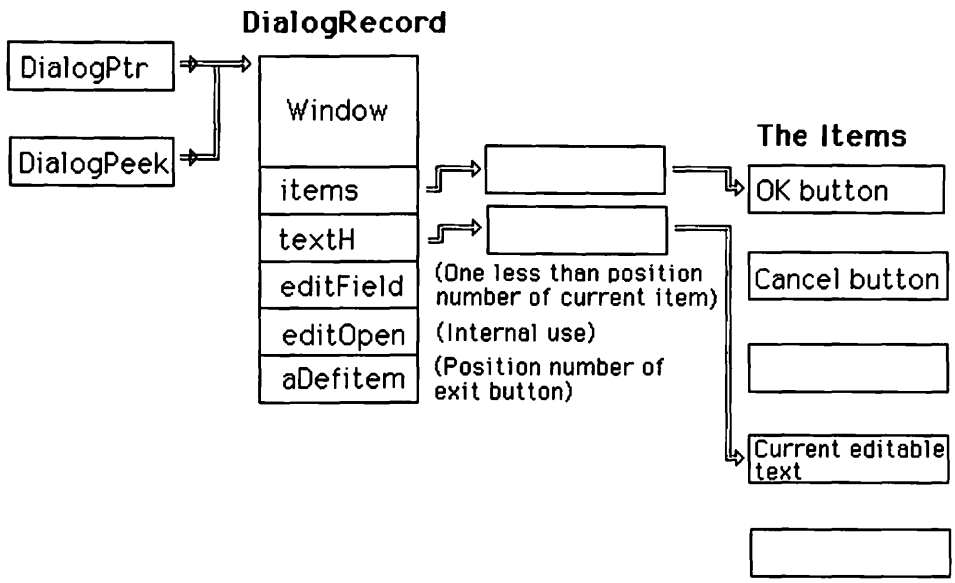
Functions and Procedures

The functions and procedures in this example program include initializing the various managers, setting default values for buttons and text, setting up the dialog, and running the dialog.

Setting up the System

The procedure "SetUpSys" initializes QuickDraw, the Font Manager, the Window Manager, the Event Manager, Text Edit, and the Dialog Manager. The routines "TEInit" to initialize Text Edit and "InitDialogs" to initialize the Dialog Manager are new. We pass a "NIL" pointer to "InitDialogs" to indicate that we want the standard error alert when the machine "bombs out".

Figure 8-7. Dialog Record Structure



Besides initializing managers, the "SetUpSyS" procedure also calls "SetDAFont" to tell the Dialog Manager to use font number one for its text items. This includes both editable text items and "static" text items that hold messages. It does not include labels for buttons, which always use the system font.

The "SetDAFont" routine expects only one parameter, an integer that specifies the font number.

Setting the Default Parameters

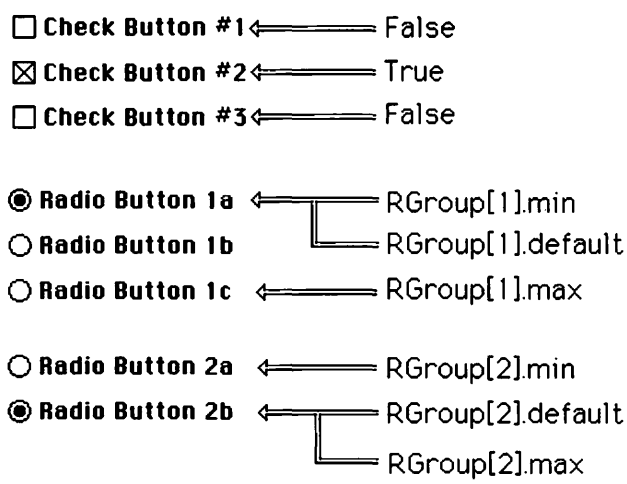
The "SetDefaults" procedure sets the original default setting for the controls and text items in the dialog.

The procedure begins by setting "theText" equal to the empty string. This is later placed in the dialog as the current value of the editable text in the text box.

The procedure initializes the Boolean "CArray" for the check buttons as FALSE, TRUE, FALSE. That is, the first check button is off, the second is on, and third is off.

Finally, the procedure initializes the minimum, maximum, and default position values for the groups of radio buttons (see Figure 8-8). For the first group, the minimum position value is given as radBtn1a, the first radio button; the maximum position value is given as radBtn1c, the last radio button. The default value is radBtn1a. For the second group, the

Figure 8-8. Setting Minimum, Maximum, and Default Values



minimum position value is given as radBtn2a, the first radio button; the maximum position value is given as radBtn2b, the last radio button. The default value is radBtn2b. The minimum and maximum values are determined by how the items are arranged in our list. The default values are chosen by the programmer.

Getting a Control Handle

Next, function "CtlHdl" gets the control handle associated with a given item. It expects a single parameter that is the position number of the item in the dialog list.

The function first calls "GetDItem" to find the vital statistics of the item. This routine returns both the type code for the item and a handle to it.

"GetDItem" expects four parameters. The first two are passed to the Dialog Manager by value. The second two are passed from the Dialog Manager by reference. The first parameter is a dialog pointer that points to the current dialog. The second parameter is an integer that holds the desired item number in the dialog's resource list. The third is an integer that holds the returned type code. The fourth is a handle to access the item's data.

Our "CtlHdl" function finishes by coercing the returned handle from type "Handle" to type "ControlHandle", placing the result in "CtlHdl" for return as a Pascal function.

Setting up Dialogs

The "SetUpDialog" function initializes our dialog, making it active on the screen. It has two local variables, "I" and "J", for loop control within the procedure.

Our procedure begins by calling the Dialog Manager's "GetNewDialog". This routine fetches the dialog and its list of resources from the resource file and allocates storage for them in memory. This step makes the dialog's window appear on the screen if it is declared *visible* in the resource definition file.

The "GetNewDialog" routine is similar to the "GetNewWindow" routine. It expects the same types of parameters and performs a similar function. In particular, the "GetNewDialog" routine expects three parameters. The first parameter is the resource identification number of the dialog. We choose a value of 1000 to correspond to our resource definition for the dialog (given later). The second parameter is a pointer to our storage for the dialog record data. We pass a NIL pointer here to indicate that the

Macintosh is to find a place for it on the heap. The third parameter is a pointer to indicate the dialog window's "visual priority". We send the expression POINTER(-1) to indicate that it should first appear in front of all other windows.

After fetching the dialog from the resource file, our procedure places the current default values into the check buttons (into their control values). Here, a FOR loop is used around the "SetCtlValue" routine. We use our own function, "CtlHdl", to convert the item number into a control handle. We use the expression:

```
ORD(CArray[I-chkBtn1+1])
```

to convert the Boolean value stored in "CArray" to a long integer. Here "ORD" converts FALSE to a numerical value of zero and TRUE to a numerical value of one.

Next, our procedure sets up the radio buttons. For each group of radio buttons, we set the reference values equal to the group number (one for the first group of radio buttons, two for the second group of radio buttons). The control value is set to zero for all buttons in the group except the default button, whose value is set to one. We use the expression:

```
ORD(J = RGroup[I].default)
```

to compute the numerical value to place in each control value.

Next, our procedure initializes the editable text. It calls "SetIText" to pass the text stored in "theText" (a string) to the "EdTxt" item belonging to the dialog.

The "SetIText" routine expects two parameters: a handle that leads to the text item in the dialog, and a string that contains the text. In our case, the handle is computed by calling our "CtlHdl" function, then coercing the resulting control handle back to type "Handle".

We call "SelIText" to move the insertion point to the end of the string. The "SelIText" routine expects four parameters. The first parameter is a dialog pointer that specifies the dialog. The second parameter is the item number of the text edit. The third and fourth parameters are integers that delimit the selection range of the text. If they are equal (as here), their common value determines the position of a single insertion point in the text. In our case, we use the "length" function to get the position of the end of the string.

Resource Definitions

Now let's look at the resource definitions for our dialog and its associated list of items.

The dialog is defined as follows:

```
Type DLOG
  ,1000
  40 85 310 425
  Visible 1 NoGoAway 0
  1000
Modal Dialog Demonstration Window
```

The first line declares that the next resource definition(s) is of type "DLOG".

The second line begins the definition of our dialog. It gives its resource identification number, which in this case is 1000.

The second line of the dialog's definition gives global coordinates for the corners of its window. You can determine the size and position of your dialogs by sketching a picture of them on paper or on the screen using a program such as MacPaint. However, you might have to fine tune the results by adjusting the numbers in the resource definition file as you develop the program.

The third line of the dialog's definition (our fourth line) gives its visibility when first fetched, the resource identification number of its definition procedure, whether it has a "goAway" box, and the value initially stored in its reference field. In our case, we want the dialog to be visible, we want to draw a standard dialog box, we want to have no "goAway" box, and we need no particular value for the reference field.

The next line gives the resource identification number of the dialog's list of controls and other items. We choose the value 1000, the same as the identification number of the dialog. Since the dialog definition and the dialog list are of different resource types, there is no conflict. In fact, it makes good sense to use the same number.

The final line is optional. It gives a title that is never displayed on the Macintosh screen but merely serves to document the dialog definition in the resource definition file.

Let's examine the dialog list definition:

```
Type DITL
  ,1000
  15
```

```
  BtnItem Enabled
  205 10 225 100
OK Button

  BtnItem Enabled
  235 10 255 100
Cancel

  StatText Disabled
  10 10 30 330
Modal Dialog Demonstration      Type "quit" to exit.

  EditText Enabled
  40 10 104 330

  BtnItem Enabled
  195 120 215 200
Cut

  BtnItem Enabled
  220 120 240 200
Copy

  BtnItem Enabled
  245 120 265 200
Paste

  ChkItem Enabled
  120 10 144 150
Check Button #1

  ChkItem Enabled
  144 10 168 150
Check Button #2

  ChkItem Enabled
  168 10 192 150
Check Button #3

  RadioItem Enabled
  120 210 144 350
Radio Button 1a

  RadioItem Enabled
  144 210 168 350
Radio Button 1b

  RadioItem Enabled
  168 210 192 350
Radio Button 1c
```

```
  RadioItem Enabled
  208 210 232 350
Radio Button 2a

  RadioItem Enabled
  232 210 256 350
Radio Button 2b
```

The first line declares that subsequent definitions are of resource type "DITL", which stands for Dialog ITem List.

The definition opens with its resource identification number, followed on the next line by the number of items in the list.

Each item is listed on three lines, with a blank line separating items. On the first line of each item definition is the item type and whether the item is enabled. The types are listed in Table 8-2. Disabled items cannot be selected, enabled items can.

The second line of each item's definition gives the position in local coordinates of its display box. Again, you can design the dialog box on paper to get these coordinates, then "fine tune" them as you test the program.

The third line gives the dialog's title, message, or, in some cases, the resource identification number.

In our case, we have fifteen items in our dialog item list: five standard buttons (type "BtnItem"), three check boxes (type "ChkItem"), five radio buttons (type "RadioItem"), one message (type "StatText"), and one editable text item (type "EditText").

Table 8-2. Dialog Item Types

Item Type in Resource Definition	*Code (Add 128 to disable)*
UserItem	0
BtnItem	4
ChkItem	5
RadioItem	6
ResCItem	7
StatText	8
EditText	16
IconItem	64

Setting Standard Buttons

The procedure "SetStdBtn" is designed to handle "standard" buttons selected in the dialog. In our program, these include the OK button, the cancel button, the cut button, the copy button, and the paste button. It is called from the "DoDialog" procedure.

Our "SetStdBtn" procedure expects one parameter, "theItem", an integer that specifies the item number of a selected standard button. It calls the appropriate dialog edit routines: "DlgCut", "DlgCopy", or "DlgPaste", depending on the value contained in "theItem". If "theItem" indicates the OK button or the cancel button, the procedure does nothing. Later, we discuss how these buttons are handled.

Setting Check Boxes

The procedure "SetChkBox" is designed to handle check box items selected in the dialog. It is called from our "DoDialog" procedure.

The procedure expects one parameter, "theItem", an integer specifying the item number of the selected check button. This procedure calls "SysBeep" to make a quick "beep" sound, then calls "GetCtlValue" and "SetCtlValue" to perform a "complement" action on the value stored in the button's control. This complement switches zero values to one values and vice versa, using the formula:

```
x := 1 - x
```

Setting Radio Buttons

The procedure "SetRadBtn" is designed to handle radio buttons selected in the dialog. It is called from the "DoDialog" procedure.

Like the previous two procedures, it expects a single parameter, "theItem", an integer specifying the item number of the selected button. The procedure begins by calling "SysBeep" to make a beep. Then it calls "GetCRefCon" to get the reference value of the button. This gives the group number of the radio button. We scan through the buttons in this group, setting all control values to zero except the one selected. We use the expression:

```
ORD(J = theItem)
```

to set these values.

Updating the Defaults

The "UpdateDefaults" procedure updates the default variables for the check buttons, radio buttons, and editable text when the OK button is pressed in the dialog. It is called from our "DoDialog" procedure.

Its two local variables, "I" and "J", are integers used as indices in loops.

The "UpdateDefault" procedure begins by updating the Boolean values for the check boxes. It checks each check box for a nonzero value, placing TRUE in the corresponding entry of CArray if it finds such a value and FALSE if not.

The procedure then uses a double FOR loop to run through each radio button in each group, setting the RGroup default value equal to the item number of the radio buttons whose control values are equal to one.

Finally, the procedure calls "GetIText" to load the text from the editable text item into the string "theText".

Doing the Dialog

The procedure "DoDialog" runs the dialog. It begins by calling "FlushEvents" to empty the event queue. Then it has a REPEAT loop, where most of the work is done.

The REPEAT loop begins by calling the Dialog Manager's "ModalDialog" routine. This routine performs several tasks. It calls the event manager, automatically draws the controls, and *tracks* the mouse. *Tracking the mouse* consists of monitoring the position of the mouse and reporting where the mouse cursor is whenever the user releases the mouse button. The "ModalDialog" routine relieves us of most of the work except for setting the control values and our own variables accordingly.

The "ModalDialog" routine expects two parameters. The first parameter is a procedure pointer to a procedure that can perform customized tracking. This is called a *filter* procedure. In our case, we place a NIL pointer here to indicate that we want the standard filter procedure provided by the Dialog Manager. The second parameter is an integer passed by reference that indicates which item was selected.

After the "ModalDialog" routine, we sort out which item was selected using a CASE statement driven by "itemHit". The cases are ranges, a special feature of Lisa Pascal. For the range:

```
cutBtn..pasteBtn
```

we call our "SetStdBtn" procedure, since these items are standard buttons. For the range:

```
chkBtn1..chkBtn3
```

we call our "SetChkBox" procedure, since these items are check boxes. And for the range:

```
radBtn1a..radBtn2b
```

we call our "SetRadBtn" procedure, since these items are radio buttons.

The REPEAT loop continues until "itemHit" indicates that the OK button or the cancel button was selected. Pressing the Return or Enter key has the same effect as selecting the OK button. This is a feature of the standard filter procedure for dialogs. In its instructions to developers, Apple encourages applications programmers to include this feature in all custom dialog filter routines. Thus, a user always can finish entering text by hitting Enter or Return, as with most computer programs.

After the REPEAT loop, we check for the OK case (OK button, Enter key, or Return key). If it is true, we call "UpdateDefaults" to set the new default values for the buttons and text.

Finally, we set "done" equal to the truth value of the statement:

```
theText = 'quit'
```

The Main Program

The main program is very short and clearly indicates the overall simple structure of the program. It begins by calling "SetUpSys", "SetDefaults", and "InitCursor" to initialize the various managers, the default variables, and the cursor.

Next, a REPEAT loop continues as long as the user does not enter the word "quit", which sets "done" to TRUE.

Within the REPEAT loop, we call "SetUpDialog" to allocate room for and initialize the dialog and its controls. We call "DoDialog" to track the action and set our variables accordingly. Next, we call "DisposDialog" to erase the dialog from memory and from the screen. If "done" is true, we call "StopAlert" to display an alert, warning that we are exiting the program. If "done" is false, we call "NoteAlert" to present a message explaining that we can now use any values generated by the dialog. In a real application, you would insert your own routine to use the data here.

These two alert calls belong to a class of four alert calls: "Alert", "StopAlert", "NoteAlert", and "CautionAlert". In all four cases, an alert is put into action. The last three cases are distinguished by special system icons displayed in the alert box (see Figure 8-9).

Each of these alert routines expects two parameters: an integer that is its resource identification number and a procedure pointer that points to a "filter" procedure to provide custom service for handling the alert. These routines return a function value that is the item number of the item selected.

Custom filters are advanced topics. In our program, we pass NIL both times to indicate that we want to use the standard service routine provided by the system. This means that we can respond only by selecting the OK button for both of our alerts. Otherwise, we must write and insert our own filter procedure. In Appendix C, we use such a filter to intercept disk insertion events which are not handled by the standard filter.

Alert Resource Definitions

Associated with each alert is a resource definition. The alert definition for our stop alert is:

```
Type ALRT
  ,1001
  100 70 200 440
  1001
  7654
```

The first line identifies the alert as resource type "ALRT". The second line specifies the resource identification number. The third line gives the global coordinates of the position of the alert's window. The fourth line gives the resource identification number of the alert's dialog item list. The last line is a hexadecimal number describing the alert's staging information.

Each alert has four *stages*, allowing the alert to execute differently in appearance, action, and sound according to how many times it has been

Figure 8-9. System Alert Icons

called by the user. The first time the user causes an alert to be invoked, it behaves according to the rules for stage one, the second time it behaves according to the rules for stage two, and so on. However, all alerts beyond stage four behave as stage four.

For each stage, we can specify the type of sound issued, whether the alert box is drawn, and the choice of default button (OK or cancel). Figure 8-10 shows how this works.

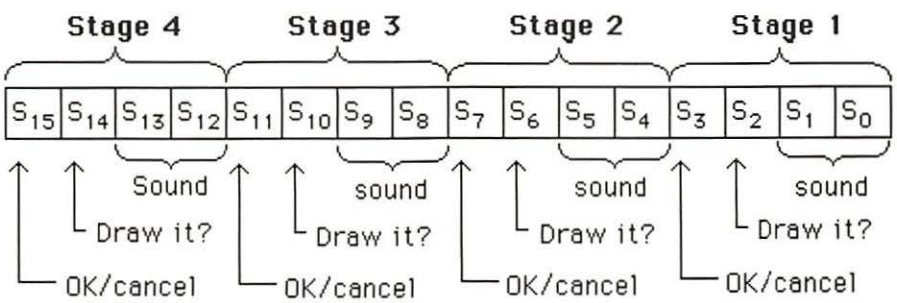
Here is the dialog item list definition for the stop alert. It uses a standard button for the OK button and "StatText" for the "Press OK to exit program." message.

```
Type DITL
  ,1001
  2

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 150 90 360
Press OK to exit the program.
```

Here is the resource definition for the note alert. It has a description almost identical to that of the stop alert.

```
Type ALRT
  ,1002
  100 70 200 440
  1002
  7654
```

Figure 8-10. Encoding Alert Stages



Here are the dialog item list definitions for the note alert. They use a standard button for the OK button and "StatText" for messages. The first message reads "Now an application can use the information from the dialog." Recall that this alert appears right after the main dialog concludes. This message indicates that the programmer can replace this alert by a routine that uses the information given by the user during the dialog.

The second message indicates that the user can press the "OK" button to return to the main dialog.

```
Type DITL
  ,1002
  3

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 150 50 360
Now an application can use the information from the dialog.

  StatText Disabled
  50 150 90 380
Press OK to return to the dialog.
```

Summary

In this chapter, we have examined a higher level of the Macintosh's built-in software. The example program is shorter than most, yet does as much or more.

We have seen how only one routine, "ModalDialog", allows us to give most of the tracking and drawing functions to the Macintosh, so that we can concentrate on *using* user-supplied information, rather than spending a lot of effort *gathering* it.

The following ROM routines were covered in this chapter:

DL-SetDAFont [Pascal only]

DL-GetDItem

DL-GetNewDialog

DL-SetIText

DL-SelIText

DL-DlgCut [Pascal only]

DL-DlgCopy [Pascal only]

DL-DlgPaste [Pascal only]
DL-GetItext
DL-ModalDialog
DL-DisposDialog
DL-StopAlert
DL-NoteAlert

9

Menus

This chapter covers the following new concepts:

- **Using Menus**
- **Menu Structure**
- **Menu Display**
- **Menu Selection**
- **Menu Resources**
- **Desk Accessories**
- **QuickDraw Shapes and Pen Attributes**

Macintosh users are familiar with "pull-down" menus and know how to use them in applications programs to select options. In this chapter, we investigate how these menus work and how to make them work for us.

Menus are managed by the Macintosh's *Menu Manager*, which, like the other managers, consists of routines and data structures in the Macintosh's memory. The Menu Manager calls on QuickDraw to draw its menus and menu bars, calls on the Resource Manager to help define its menus, and calls on the Event Manager while tracking the mouse for menu selection.

Following a discussion of menus and the Menu Manager, we present an example program that demonstrates how to use the Menu Manager's routines to set up and track menus. This example also shows how an applications program can use the selection information returned from the Menu Manager.

Menus

Menus are lists of options available to the user. Each application has a different set of menus because it has a different set of actions. For example, in this chapter's program, our actions focus on drawing various shapes and selecting how they are drawn. A special "Apple" menu also allows us to select various desk accessories.

We see how menus appear to the user, then examine their data structures and how to program them.

How Menus Appear to the User

At the top of the screen, in the topmost 20 rows of pixels, sits the *menu bar*, which provides access to an application's active menus. The title of each active menu appears in the menu bar.

Normally, no windows are allowed to overlap this area of the screen. In particular, the initial positions of all windows should be defined so that they avoid the menu bar. The drag bounds should also be set for subsequent window positions so that the windows can never be dragged into the menu bar.

When the mouse cursor is pressed in the title area of a menu, the menu should be immediately displayed underneath its title. The menu itself may be wider than its title, overlapping space that is occupied by adjacent menus when they appear. However, only one menu appears at a time, though all titles are visible at once (see Figure 9-1).

Once a menu is selected, the user can run the mouse cursor down the menu as the mouse button is pressed down, highlighting each item as the cursor passes over it. When the mouse button is released over an item, that item is selected, and control returns to the applications program.

The entire menu selection process, from the time the mouse button is pressed until it is released, is controlled by one Menu Manager routine called "MenuSelect". We see how this routine is invoked when we study our example program.

Menu Lists and Structures

For most purposes, you do not have to know how the Menu Manager stores information about menus. You only have to call the appropriate Menu Manager routine; the Menu Manager takes care of the details. However, if you are interested in how the Menu Manager works, you must first understand its data structures. This is the only way to unlock the Macintosh's "hidden powers" to understand the full capabilities of the Menu Manager,

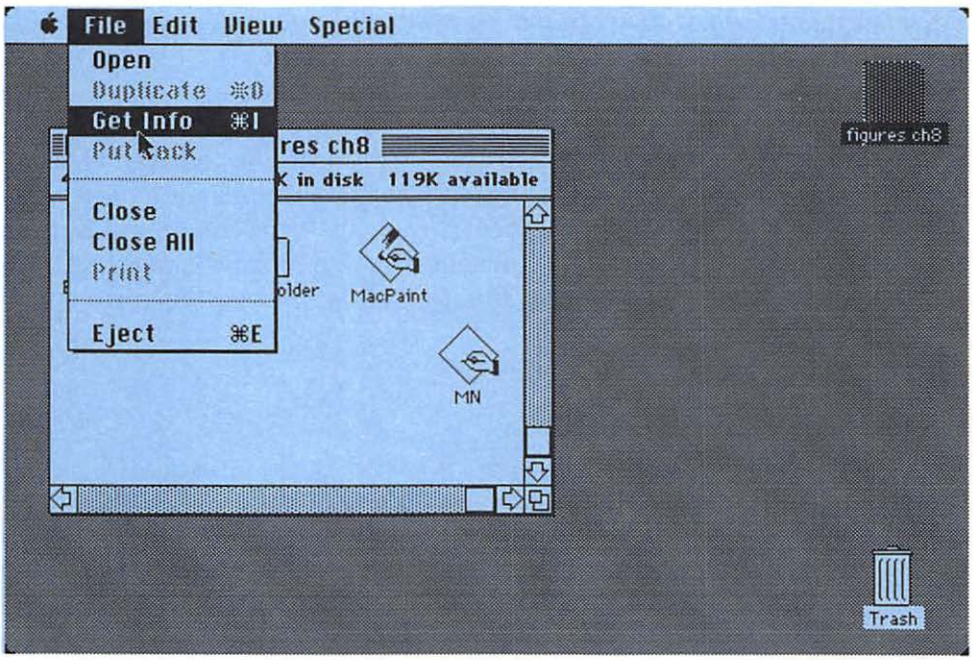
to define your custom menus, and to answer difficult questions that could arise while using a debugger. We don't set up custom menus in this book. However, reading the following menu descriptions should provide a good foundation for doing so. Details are contained in Apple's *Inside the Apple Macintosh*.

The Menu Manager stores all menus under its control in a master list of menus. The system variable "MenuList", stored in the low memory of the Macintosh, contains a handle to this list. The list contains six bytes for each menu. The first four bytes contain a handle to the menu's data structure, and the last two bytes contain the horizontal position of the menu's title in the title bar (see Figure 9-2).

Each menu is accessed through a handle of type "MenuHandle". This points to a pointer of type "MenuPtr", which points to the actual data. Here is the Pascal declaration for the menu data structure:

```
MenuHandle = ^MenuPtr;
MenuPtr    = ^MenuInfo;
MenuInfo = RECORD
             menuID:      INTEGER;
```

Figure 9-1. A Menu is Selected



```
    menuWidth:   INTEGER;
    menuHeight:  INTEGER;
    menuProc:    Handle;
    enableFlags: PACKED ARRAY [0..31] OF BOOLEAN;
    menuData:    Str255;
END;
```

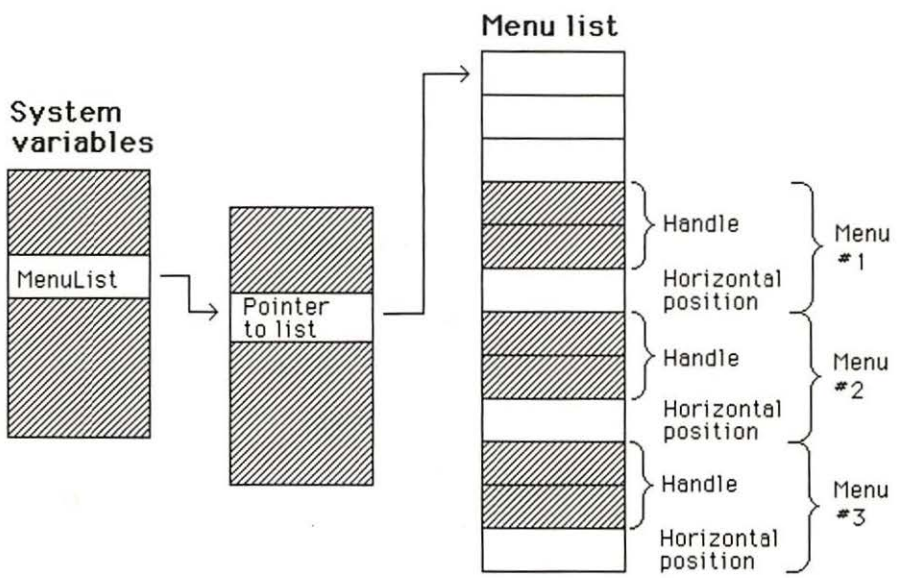
This "MenuInfo" structure contains the data for managing a single menu, including its size, the name and style of each item, and the location of the procedure to draw and track the menu.

The ".menuID" field contains an integer assigned by the programmer to the menu when it is first defined. Thereafter, it can be used by the programmer to reference the menu and is used by the Menu Manager to specify the selected menu from the menu selection process.

The ".menuWidth" field contains the width in pixels and the ".menuHeight" field contains the height in pixels of the menu.

The ".menuProc" field contains a handle to the menu definition procedure. This procedure draws the menu, determines its size, and performs tracking functions during the menu selection process. Programmers can insert their own menu definition procedures in this field, thereby customizing their own nonstandard types of menus. However, this is

Figure 9-2. The Master List of Menus



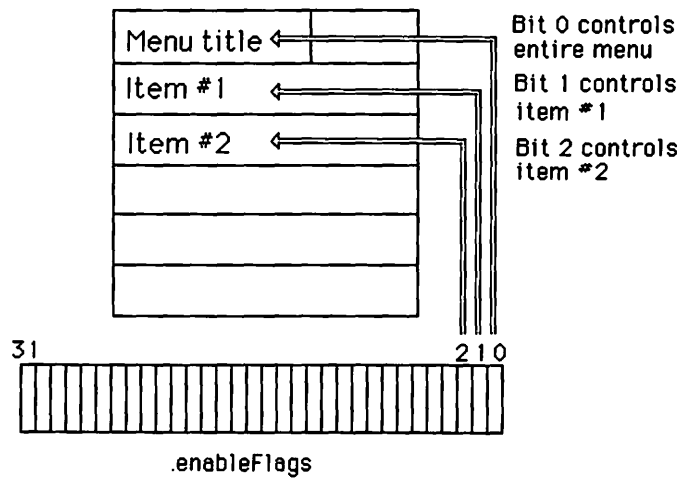
beyond the scope of this book. To define your menus, consult Apple's *Inside the Apple Macintosh*.

The ".enableFlags" field contains 32 Boolean variables, one for the entire menu and one for each entry (see Figure 9-3). The first Boolean determines if the menu itself is enabled or disabled, as do subsequent Booleans for each item in the menu. In standard menus, "disabled" means that the item's name appears in light gray rather than black and cannot be selected by the user. When defining your custom menus, you can indicate the enable/disable status of menu items in any way.

As far as Pascal is concerned, the ".menuData" field is a string. That string contains the menu's title, followed by the title of each item in the string and such information as the style for each item, key equivalents, any marks in front of items, and any icons associated with these menu items.

Again, it is usually not necessary for a programmer to directly access the fields of a menu's record structure. A number of Menu Manager routines change such features as items' titles, styles, check marks, and enable status. A programmer may want to read a menu's title from the last field of its record structure, but he or she should never modify the menu's title as it sits in the menu data string. This could destroy the remaining menu data.

Figure 9-3. Enable Flags



The Example Program

The example program in this chapter demonstrates how to manage menus. It also illustrates how to manage desk accessories such as the scrapbook, alarm clock, note pad, and calculator, which are treated as system tasks. In addition, it allows you to quickly explore a number of QuickDraw shapes and drawing attributes (see Figure 9-4).

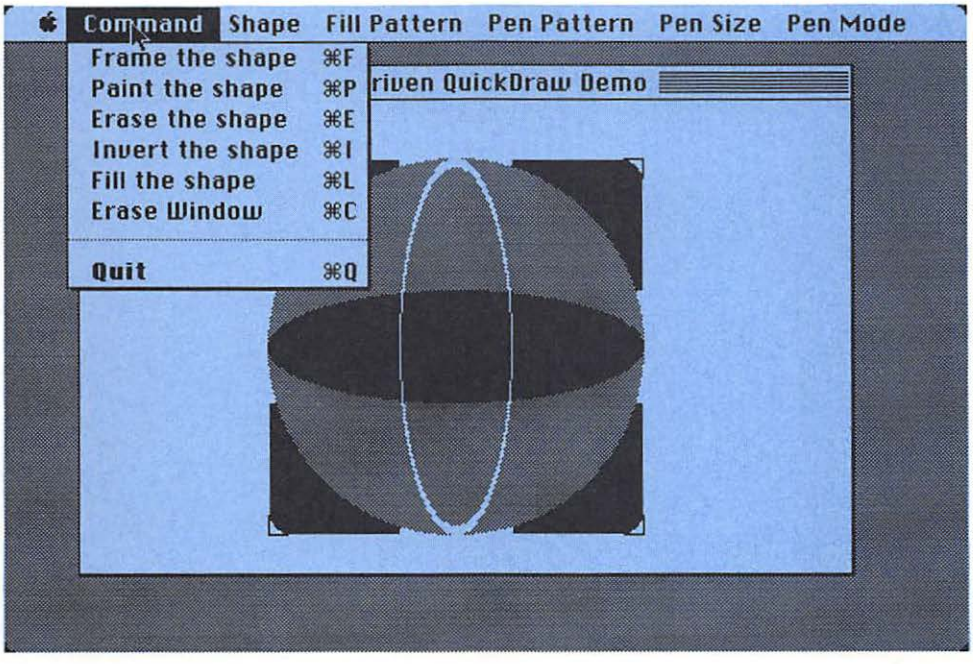
This program allows you to create standard shapes — rectangles, ovals, and rounded rectangles — that can be drawn with various shades of gray using framing, filling, painting, erasing, and inverting. Appropriately, the program is "menu-driven", demonstrating how menus can be used to select options and commands.

Now let's look at the program in detail. When the program starts, it displays a blank window and a full menu bar. The window is entitled "Menu Driven QuickDraw Demo". The menu bar contains titles for the following seven menus:

Apple menu

Command menu

Figure 9-4. Our Menu Program



Shape menu

Fill Pattern menu

Pen Pattern menu

Pen Size menu

Pen mode

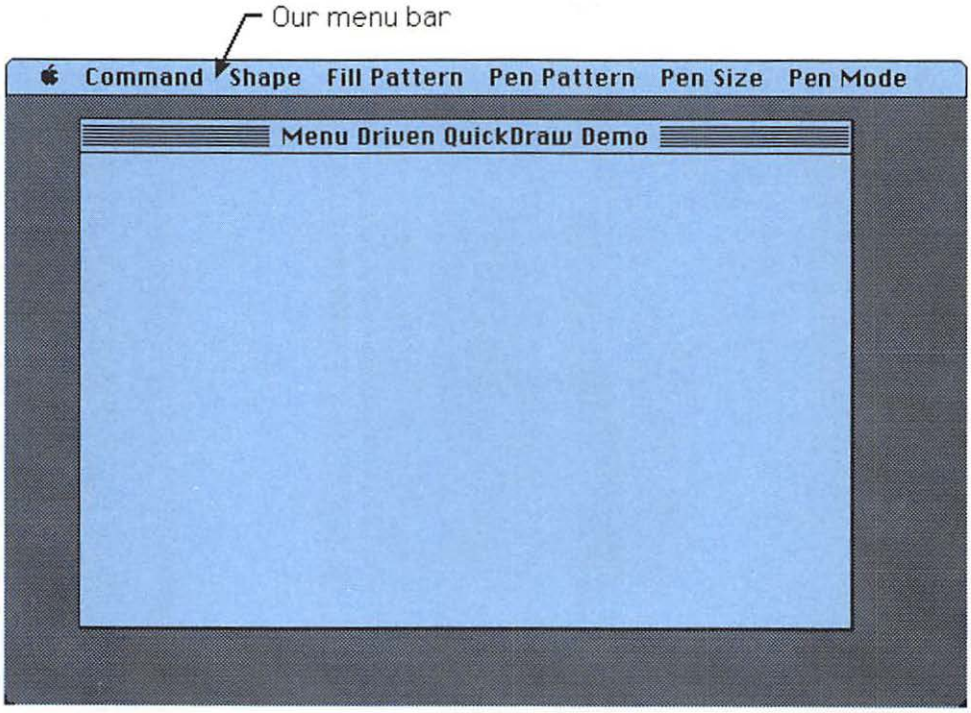
across the top of the screen (see Figure 9-5).

You can examine the contents of a given menu by pressing the mouse button in the appropriate region of the menu bar. This pulls the menu down, making it appear in front of everything on the screen except the cursor (see Figure 9-6).

The Apple menu is divided into two parts. One part contains a special "About Menu" entry, the other part contains the titles of the desk accessories (see Figure 9-7).

To select an item in the Apple menu, pull it down, move the mouse button down until the desired item is selected, then release the mouse

Figure 9-5. Our Menu Bar



button. A single call in our program makes this entire process happen automatically.

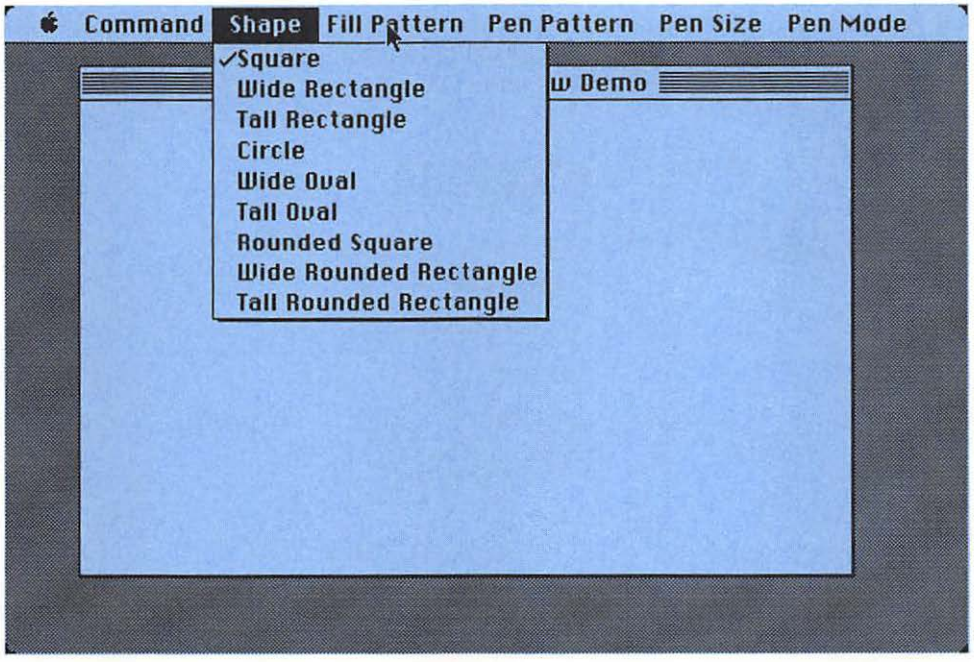
The Command menu selects an action to be performed immediately (see Figure 9-8). These include framing, painting, erasing, inverting, and filling a shape, as well as erasing the window's contents and quitting the program. The option of quitting the program is separated from other options and written in boldface.

The remaining menus act differently than the first two menus. They use check marks to set parameters rather than directly cause an action. Whenever you pull down one of these menus, you see that the currently selected item is checked. If you select another item, then that item is checked the next time that you pull down the menu.

The Shape menu is used to select the particular shape that is drawn (see Figure 9-8). The available shapes are square, wide rectangle, and tall rectangle, circle, wide oval, tall oval, rounded square, wide rounded rectangle, and tall rounded rectangle. Initially, the square is selected.

The Fill Pattern menu selects the pattern that is used when the shape is filled. The available fill patterns are the QuickDraw default patterns

Figure 9-6. Selecting a Shape



White, Black, Gray, LtGray, and DkGray. Initially, the Fill Pattern is set to Gray.

The Pen Pattern menu selects the pattern for framing and painting objects. Its available patterns are the same as for the Fill Pattern menu. Initially, it is set to Black.

The Pen Size menu offers a range of pen sizes including 1 by 1, 1 by 5, 5 by 1, 5 by 5, and 10 by 10. The initial size is 1 by 1.

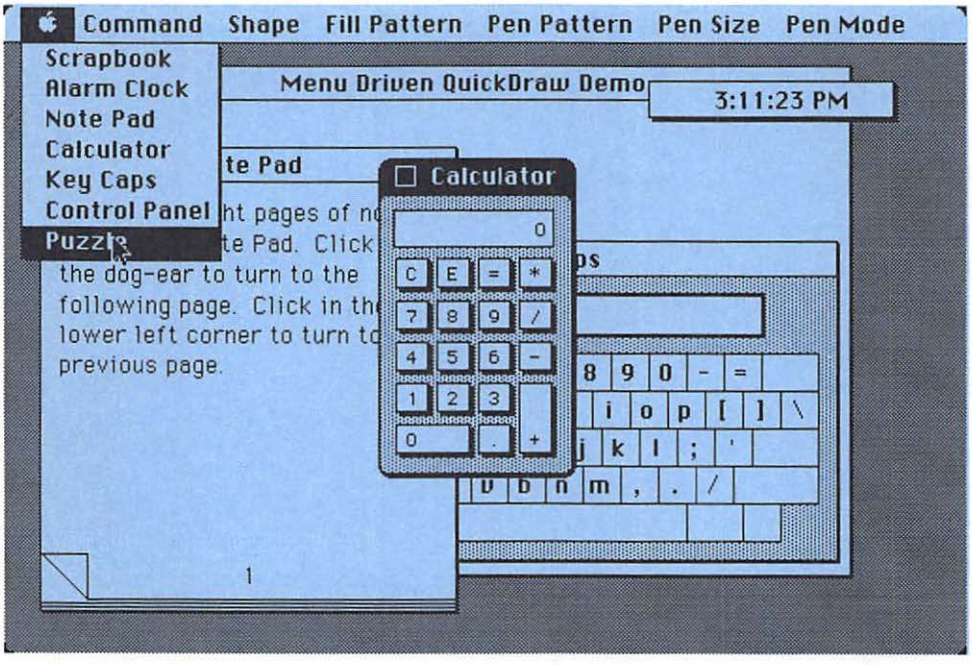
The Pen Mode menu selects one of the pen modes — Copy, Or, Xor, Bic, NotCopy, NotOr, NotXor, and NotBic (see Figure 9-9). The initial mode is Copy. These are the eight pen patterns available to the programmer. They determine how the bits in the pen pattern are combined with what is on the screen. Note that the pen modes work with "Paint" commands but not "Fill" commands.

Let's review how these eight pen modes work and what they do.

In the Copy mode, the pattern is "copied" directly to the screen, overwriting what was there. This is the default pen mode (see Table 9-1).

In the Or mode, the bit values in the pattern are Ored with what was on the screen (see Table 9-2). This tends to overstrike what was on the screen.

Figure 9-7. Selecting a Desk Accessory



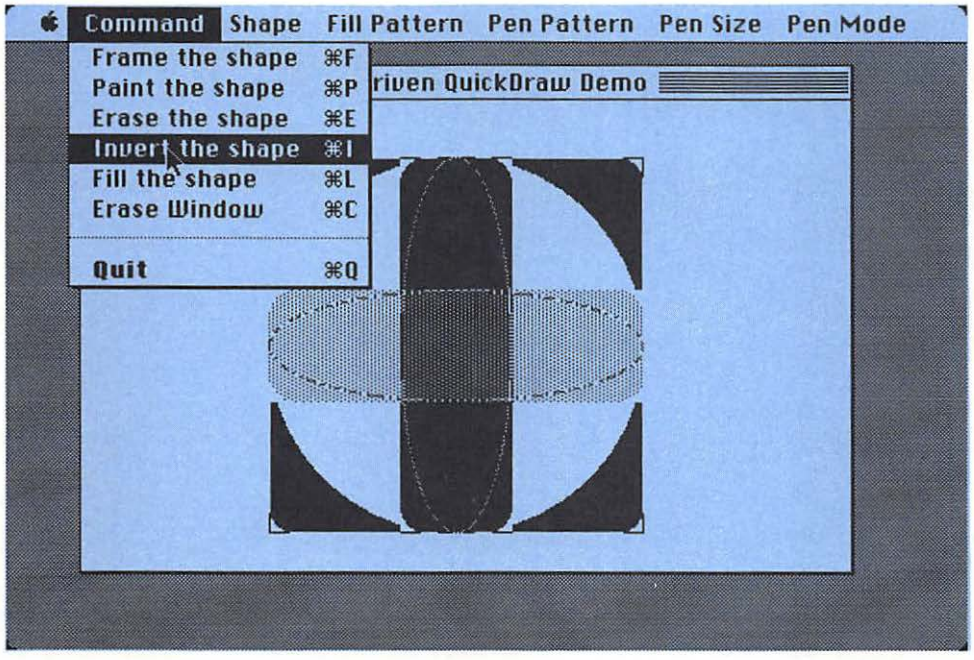
In the Xor mode, the bits in the pattern are Xored with what is on the screen (see Table 9-3). This is useful for creating cursorlike objects, since it reverses the bits on screen that are equal to one in the pattern and leaves other bits alone. When the same pattern is Xored a second time to the same place on the screen, the bits in the screen return to their previous state. To move a cursor, Xor it to its current position, thus erasing it; then Xor it to its new position, making it appear there.

In the Bic mode, the bits in the pattern are combined with those on the screen using the bit clear operation (see Table 9-4). All bits on the screen that correspond to bits in the pattern (set equal to one) are cleared (set equal to zero). This is useful to selectively erase portions of the screen.

In the four Not modes, the pattern is inverted as it is applied to the screen using one of the four Not operations. These options are not discussed here.

When a drawing command is selected from the Command menu, drawing occurs only in our display window. This window can be dragged but not resized or scrolled. Resizing can be added without too much difficulty. However, scrolling is harder, since the picture is composed of a potentially long sequence of drawing commands; therefore, updating the

Figure 9-8. Selecting a Command



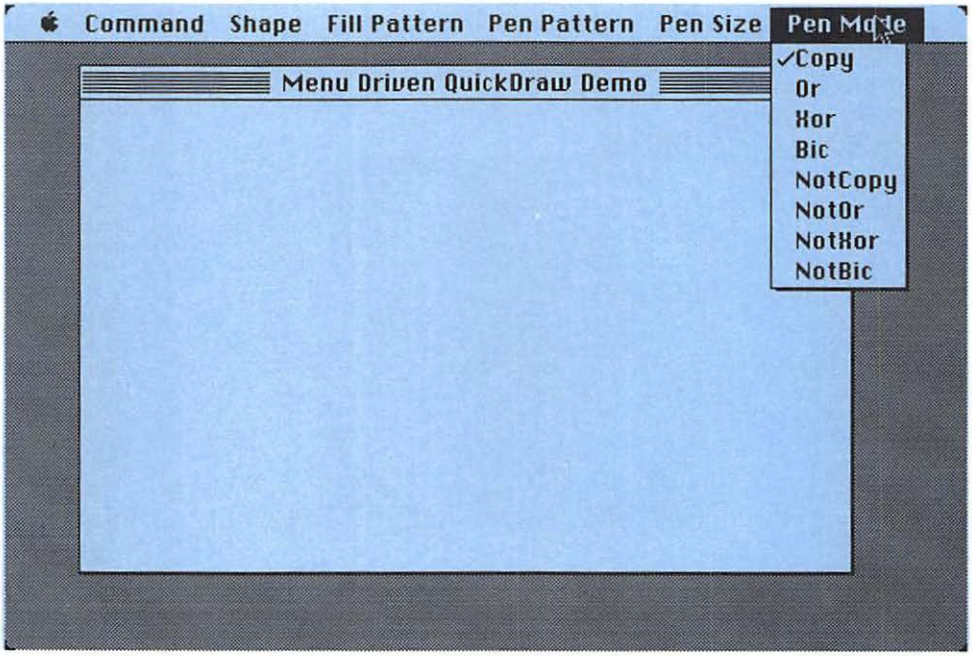
picture requires special attention. As it is, the program does not try to properly update the picture. This is evident if you move desk accessories in front of the display menu.

Experiment with this program, exploring its various modes, shapes, sizes, and patterns.

Here is the program:

```
PROGRAM MenuDemo;
    {$R-}{$X-}

    USES
        {$U Obj/MemTypes     } MemTypes,
        {$U Obj/QuickDraw    } QuickDraw,
        {$U Obj/OSIntf       } OSIntf,
        {$U Obj/ToolIntf     } ToolIntf;

    CONST
        {menu IDs}
        AppleMenu    = 1000;    { desk accessory menu}
        ComMenu      = 1001;    { Command menu}
        ShapeMenu    = 1002;    { Shape menu}
```

Figure 9-9. Selecting a Pen Mode



```
    FillPatMenu = 1003;     { Fill Pattern menu}
    PenPatMenu  = 1004;     { Pen Pattern menu}
    PenSizeMenu = 1005;     { Pen Size menu}
    PenModeMenu = 1006;     { Pen Mode menu}
    lastMenu = 7;           { number of menus}

VAR
    done  : BOOLEAN;
    theEvt: EventRecord;
    mainWindow, theWindow: WindowPtr;
    dragBnds: Rect;
    myMenus: ARRAY [1..lastMenu] OF MenuHandle;
    theShape, theFillPat, thePenPat, thePenSize, thePenMode : INTEGER;

PROCEDURE SetUpSys;
    BEGIN
        InitGraf(@thePort);
        InitFonts;
        InitWindows;
```

Table 9-1. The Copy Mode

Result = New		
Originally on Screen	*New Pattern*	*Result*
0	0	0
0	1	1
1	0	0
1	1	1

Table 9-2. The Or Mode

Result = Original OR New Pattern		
Originally on Screen	*New Pattern*	*Result*
0	0	0
0	1	1
1	0	1
1	1	1

```
    InitDialogs(NIL);
    TEInit;
    SetEventMask(everyEvent);
    FlushEvents(everyEvent,0);
    InitCursor;

    {Define bounds for dragging the window}
    WITH screenBits.Bounds DO
       SetRect(dragBnds,left+4,top+24,right-4,bottom-4);

    mainWindow := GetNewWindow(256,NIL,POINTER(-1));
    done := FALSE;
END;

PROCEDURE ClickItem(menuIndex,theItem: INTEGER;
                    VAR itemNum: INTEGER);
BEGIN
    CheckItem(myMenus[menuIndex],itemNum,FALSE);
    itemNum := theItem;
```

Table 9-3. The XOR Mode

Result = Original XOR New Pattern

Originally on Screen	*New Pattern*	*Result*
0	0	0
0	1	1
1	0	1
1	1	0

Table 9-4. The BIC Mode

Result = Bit Clear Original Using New Pattern as Mask

Originally on Screen	*New Pattern*	*Result*
0	0	0
0	1	0
1	0	1
1	1	0

```
      CheckItem(myMenus[menuIndex], itemNum, TRUE);
   END;

PROCEDURE SetUpMenu;
   VAR
      I : INTEGER;
   BEGIN
      InitMenus;
      myMenus[1] := GetMenu(AppleMenu);
      myMenus[2] := GetMenu(ComMenu);
      myMenus[3] := GetMenu(ShapeMenu);
      myMenus[4] := GetMenu(FillPatMenu);
      myMenus[5] := GetMenu(PenPatMenu);
      myMenus[6] := GetMenu(PenSizeMenu);
      myMenus[7] := GetMenu(PenModeMenu);

      AddResMenu(myMenus[1], 'DRVR');
      FOR I:= 1 TO lastMenu DO InsertMenu(myMenus[I], 0);
      DrawMenuBar;

      {initial option values}
      theShape   := 1;
      theFillPat := 3;
      thePenPat  := 2;
      thePenSize := 1;
      thePenMode := 1;

      {check marks for initial option values}
      CheckItem(myMenus[3], theShape,   TRUE);
      CheckItem(myMenus[4], theFillPat, TRUE);
      CheckItem(myMenus[5], thePenPat,  TRUE);
      CheckItem(myMenus[6], thePenSize, TRUE);
      CheckItem(myMenus[7], thePenMode, TRUE);

   END; {of SetUpMenus}

PROCEDURE DrawShape(theCommand: INTEGER);
   VAR
      shapeRect, theScreen : Rect;
      curv : Point;
      dpat : Pattern;

   BEGIN
      SetPort(mainWindow);
      theScreen := thePort^.portRect;
      SetPt(curv, 30, 30);
```

```
CASE theFillPat OF
   1: dpat := White;
   2: dpat := Black;
   3: dpat := Gray;
   4: dpat := LtGray;
   5: dpat := DkGray;
   END;
CASE thePenPat OF
   1: PenPat(White);
   2: PenPat(Black);
   3: PenPat(Gray);
   4: PenPat(LtGray);
   5: PenPat(DkGray);
   END;
CASE thePenSize OF
   1: PenSize(1,1);
   2: PenSize(1,5);
   3: PenSize(5,1);
   4: PenSize(5,5);
   5: PenSize(10,10);
   END;
CASE thePenMode OF
   1: PenMode(PatCopy);
   2: PenMode(PatOr);
   3: PenMode(PatXor);
   4: PenMode(PatBic);
   5: PenMode(NotPatCopy);
   6: PenMode(NotPatOr);
   7: PenMode(NotPatXor);
   8: PenMode(NotPatBic);
   END;

CASE theShape OF
   1,4,7: SetRect(shapeRect, 100,  30, 300, 230); {Square}
   2,5,8: SetRect(shapeRect, 100, 100, 300, 160); {WideRect}
   3,6,9: SetRect(shapeRect, 170,  30, 230, 230); {TallRect};
   END; {aspect ratio}
CASE theShape OF
   1,2,3: CASE theCommand OF
             1:  FrameRect(shapeRect);
             2:  PaintRect(shapeRect);
             3:  EraseRect(shapeRect);
             4:  InvertRect(shapeRect);
             5:  FillRect(shapeRect,dpat);
             6:  EraseRect(theScreen);
             8 : done := TRUE;
             END; {of Shapes 1,2,3: rectangles}
```

```
            4,5,6: CASE theCommand OF
                        1:  FrameOval(shapeRect);
                        2:  PaintOval(shapeRect);
                        3:  EraseOval(shapeRect);
                        4:  InvertOval(shapeRect);
                        5:  FillOval(shapeRect,dpat);
                        6:  EraseRect(theScreen);
                        8 : done := TRUE;
                        END; {of Shapes 4,5,6: ovals}
            7,8,9: CASE theCommand OF
                        1:  FrameRoundRect(shapeRect,curv.h,curv.v);
                        2:  PaintRoundRect(shapeRect,curv.h,curv.v);
                        3:  EraseRoundRect(shapeRect,curv.h,curv.v);
                        4:  InvertRoundRect(shapeRect,curv.h,curv.v);
                        5:  FillRoundRect(shapeRect,curv.h,curv.v,dpat);
                        6:  EraseRect(theScreen);
                        8 : done := TRUE;
                        END; {of Shapes 7,8,9: wide rounded rectangle}
            END; {of shapes}
    END; {DrawShape}

PROCEDURE DoAppleMenu(theItem: INTEGER);
    VAR
        refNum: INTEGER;
        name:   Str255;
    BEGIN
        If theItem = 1
            THEN theItem := Alert(1001,NIL)
            ELSE
                BEGIN
                    GetItem(myMenus[1],theItem,name);
                    refNum := OpenDeskAcc(name);
                END;
    END;

PROCEDURE SelectMenu(selection : LongInt);
    VAR
        theMenu, theItem : INTEGER;
    BEGIN
        theMenu := HiWord(selection);
        theItem := LoWord(selection);
        CASE theMenu OF
            AppleMenu:    DoAppleMenu(theItem);
            ComMenu:      DrawShape(theItem);
            ShapeMenu:    ClickItem(3, theItem, theShape);
            FillPatMenu:  ClickItem(4, theItem, theFillPat);
            PenPatMenu:   ClickItem(5, theItem, thePenPat);
```

```
            PenSizeMenu: ClickItem(6, theItem, thePenSize);
            PenModeMenu: ClickItem(7, theItem, thePenMode);
            END;     {of theMenu CASE }
         HiliteMenu(0); {to unhighlight selected menu in menu bar}
      END;

BEGIN     {main program}
      SetUpSys;
      SetUpMenu;

      REPEAT
         SystemTask;
         IF GetNextEvent(everyEvent, theEvt) THEN
            CASE theEvt.what OF
               mouseDown:
                  CASE FindWindow(theEvt.where, theWindow) OF
                     inMenuBar:
                        SelectMenu(MenuSelect(theEvt.where));
                     inSysWindow:
                        SystemClick(theEvt, theWindow);
                     inDrag:
                        DragWindow(theWindow, theEvt.where, dragBnds);
                     inContent, inGrow:
                        SelectWindow(theWindow);
                     END;
               keyDown:
                  SelectMenu(MenuKey(Chr(theEvt.message MOD 256)));
               updateEvt, activateEvt:
                  BEGIN
                     theWindow := windowPtr(theEvt.message);
                     BeginUpdate(theWindow);
                     EndUpdate(theWindow);
                  END;
               END; {of what event}
      UNTIL done;
END.
```

Data Structures

Our "MenuDemo" program has the standard USES section. The CONST section defines the menu identification numbers of the menus, assigning constant identifiers for each. It also defines the constant "lastMenu" as equal to seven, the total number of menus. Using constants in this manner makes the program more readable and easier to maintain.

Global Variables

The VAR section contains a number of global variables. Some are familiar from previous programs, some are peculiar to menu management.

The first two global variables, "done" and "theEvt", have been used to manage events before. The first is a Boolean that controls the termination of the main loop. The second is an event record that holds the "what" and "where" information for our events.

The global variables "mainWindow" and "theWindow" are window pointers. The first points to our main window, which displays the shapes. The second is a general-purpose window pointer, as used in the example programs of the last few chapters.

The rectangle "dragBounds" provides the bounds for window dragging. Later, we set these bounds so that the windows never go completely off the screen or overlap the menu bar.

The global array of menu handles, "myMenus", provides access to the menus in our program. Later we will see how this array structure makes it easy to initialize all menus at once in a simple FOR loop.

Finally, the integers "theShape", "theFillPat", "thePenPat", "thePenSize", and "thePenMode" hold current choices from the last five menus. These use check mark options; that is, they show which menu items are selected by displaying check marks.

Procedures

The various procedures in this program initialize the system, initialize the menus, draw the shapes, and track the menus. Let's look at them in detail.

Initializing the System

The first procedure, "SetUpSys", initializes QuickDraw, the Font Manager, the Window Manager, the Dialog Manager, Text Edit, and the Event Manager. It also calls "TEInit" to initialize Text Edit. Though you might not need all these managers in your program, they must be initialized to ensure that the desk accessories do not crash.

"SetUpSys" has a more sophisticated way to set the window drag limits in the rectangle "dragBnds". "SetRect" (surrounded by a WITH statement) defines the drag limits relative to the screen limits as given by "screenBits.Bounds". The drag limits are set so that some portion of the window always stays within the screen and windows do not cover any

portion of the menu bar. Because we use "screenBits" to define the drag limits, our program should work even if the screen size changes.

This initialization procedure next calls "GetNewWindow" to get the window definition parameters from our resource file and draw the window. Here is the resource definition for this window:

```
Type WIND
     ,256
     Menu Driven QuickDraw Demo
     50 40 300 450
     Visible NoGoAway
     0
     0
```

Lastly, the procedure sets "done" equal to false for our event loop.

Checking Menu Items

The "ClickItem" procedure updates check marks for the last five menus. It expects three integer parameters: "menuIndex", to specify a particular menu; "theItem", to specify the new item that should be checked in that menu; and "itemNum", to specify the item that is currently checked in that menu. The procedure removes the check mark from the previously checked item, updates the variable "theItem" so that it now indicates the current choice, and places a check mark on this new choice.

The procedure begins by calling the Menu Manager routine "CheckItem" to uncheck the currently checked item. This routine expects three parameters: a menu handle that specifies a menu, an integer that specifies a particular item within that menu, and a Boolean that specifies whether the item is to be checked or unchecked. In this case, we pass "myMenus[menuIndex]" in the first parameter to indicate the menu to be modified, "itemNum" in the second parameter to indicate the currently checked item, and FALSE to the third parameter to indicate that it should be unchecked.

The procedure then updates "itemNum" by assigning the value of "theItem" to it.

Finally, the procedure checks the new choice by calling "CheckItem" with the same expressions in its first two parameters, indicating the same menu with the updated choice of item. We pass TRUE to the last parameter to indicate that we want the item checked.

Setting Up Menus

The procedure "SetUpMenus" initializes all menus for this program. It has one local variable, an integer "I", which is an index to a FOR loop.

The procedure begins by calling "InitMenus" to initialize the Menu Manager.

We call "GetMenu" seven times to get the menu descriptions for all seven menus from our resource file attached to our program. In each case, we pass an identifier defined in our CONST section. The names of these identifier constants have been chosen for ease of recognition. For example, "AppleMenu" identifies the Apple menu, "ComMenu" identifies the Command menu, and "ShapeMenu" identifies the Shape menu.

Menu identification numbers should differ from each other and should never equal zero, since zero signals a "nonchoice".

We then use a FOR loop to add these menus to the Menu Manager's master list of menus. This loop is indexed by I and runs from 1 to "lastMenu". It calls "InsertMenu" to add each menu to the list.

The "InsertMenu" routine expects two parameters: the menu handle of the menu we wish to add to the master list of menus, and an integer that specifies where in the list that menu should go. In our case, we pass zero to indicate that each menu, in turn, is added to the end of the list. In general, if this integer parameter is one of the menus already in the list, then the new menu is added before that menu.

The first menu, the Apple menu, is different from the others; it is not completely defined in the resource definition file. Only its first two entries are defined there. We must call "AddResMenu" to search the resource files for the titles and identification numbers of the system's desk accessories to add them to the Apple menu.

The "AddResMenu" routine expects two parameters: a menu handle and a resource type. We specify the menu handle as "myMenus[1]", which we have just set up for the Apple menu, and we specify 'DRVR' to indicate that the system should search for resources of the type of desk accessories.

We then call "DrawMenuBar" to display the titles of all our menus across the menu bar. The routine expects no parameters. It merely uses the Menu Manager's master list of menus to find these titles.

Finally, we use assignment statements to initialize the current values of the choices in the last five menus, then make calls to "CheckItem" to initialize the check marks in the actual menus.

Menu Resource Definitions

Let's look at how these menus are defined in the resource definition field attached to our program.

As mentioned earlier, the definition for the Apple menu is not completely contained in our resource definition file. The last part is gathered from system resources using the "AddResMenu" procedure.

The Apple menu appears as follows in the resource definition file:

```
Type MENU
  ,1000
  \14
       About Menu...
       (--------------
```

The first line after the "Type" specification contains the menu identification number, preceded by a comma. The next line contains the title; in this case, the Apple symbol. The "\14" indicates the ASCII code of this symbol. In particular, the "\" is a *meta-character* indicating that the next digits specify the hexadecimal representation of a number.

Table 9-5 lists these special meta-characters.

The first menu item, "About Menu", allows us to display an alert box with information about our program. The second menu item serves as a separator between the "About Menu" command and the commands that open desk accessories. It appears in the menu as a disabled dashed line and is indicated in the resource definition by a left parenthesis followed by the dashed line. The left parenthesis is also a meta-character. It specifies that the item is disabled. We see later how other items are filled in when the program executes.

Table 9-5. Meta-Characters for Menu Definitions

Meta-Character	*Meaning*
; or <CR>	Separates items
^	Item icon
!	Item mark
<	Item style
/	Item keyboard equivalent
(	Item is disabled

The remaining menus are defined in our resource definition file. The Command menu definition appears as follows:

```
Type MENU
  ,1001
  Command
       Frame the shape/F
       Paint the shape/P
       Erase the shape/E
       Invert the shape/I
       Fill the shape/L
       Erase Window/C
       (----------------
       Quit/Q <B
```

The first line contains the menu identification number, preceded by a comma. The second line contains the title "Command".

Each remaining line contains a title for an item in this menu. Items one through six and item eight have keyboard equivalents. These are indicated (after the title) by the meta-character slash followed by the key symbol. They are an alternative to selecting menu items. To use these key equivalents, the user must hold down the ⌘ key like a shift key while pressing the indicated key.

As an example of key equivalents, the item title "Frame the shape" is followed by "/F", which indicates that its keyboard equivalent is obtained by hitting ⌘ F.

The seventh menu item is as a separator between the QuickDraw commands and the "Quit" command. It appears in the menu as a disabled dashed line and is indicated in the resource definition by a left parenthesis followed by the dashed line. Again, the left parenthesis is a meta-character that specifies that the item is disabled.

The eighth line appears in boldface. This is indicated by the meta-character "<" followed by a "B" for boldface immediately after the title. Table 9-6 lists the other style options.

The remaining five menus are less fancy and contain no meta-characters:

```
Type MENU
 ,1002
  Shape
       Square
       Wide Rectangle
       Tall Rectangle
       Circle
       Wide Oval
```

```
     Tall Oval
     Rounded Square
     Wide Rounded Rectangle
     Tall Rounded Rectangle

,1003
Fill Pattern
    White
    Black
    Gray
    LtGray
    DkGray

,1004
Pen Pattern
    White
    Black
    Gray
    LtGray
    DkGray

,1005
Pen Size
    1 by 1
    1 by 5
    5 by 1
    5 by 5
   10 by 10

,1006
Pen Mode
    Copy
    Or
    Xor
    Bic
```

Table 9-6. Style Options for Menu Items

Symbol	*Meaning*
B	Bold
I	Italic
U	Underline
O	Outline
S	Shadow

```
NotCopy
NotOr
NotXor
NotBic
```

Drawing the Shapes

The procedure "DrawShape" draws the various shapes in our display window. It expects one parameter, which specifies a command from the Command menu.

The "DrawShapes" procedure has several local parameters: "shapeRect", a rectangle that determines the aspect ratio of the shape to be drawn; "theScreen", a rectangle that specifies the size of the screen; "curv", of type "Point", that specifies the curvature of the rounded rectangles; and "dpat", a pattern that fills the shape (as opposed to painting it).

We begin the procedure by calling "SetPort" to make our "mainWindow" into the current window. Next, we assign "thePort^.portRect" to the variable "theScreen". This simplifies erasing the contents of the window later in the program.

Next, we set "curv" equal to (30,30) to specify the curvature of the rounded rectangles.

The rest of the program sorts the selections that determine the shape and draw it.

First, we use a CASE statement to determine the fill pattern, setting "dpat" according to the selection made. Next, we use another CASE statement to determine the pen pattern, calling "PenPat" with the appropriate pattern. Then we determine the pen size with a third CASE statement, calling "PenSize" with the appropriate sizes. The pen mode is determined by a simple CASE statement, calling "penMode" with the appropriate parameters.

The shape is a more complicated matter. A CASE statement first assigns the aspect ratio of the shape to "shapeRect". For shape selections one, four, and seven, we choose a square; for shape selections two, six, and eight, we choose a wide rectangle; and for selections three, six, and nine, we choose a tall rectangle. This means, for example, that a tall oval is assigned the underlying shape of a tall rectangle, a wide oval is assigned the underlying shape of a wide rectangle, and so on.

Next, we determine the set of shape commands to draw the shape. For shape selections one, two, and three, we use rectangle commands; for shape selections four, five, and six, we use oval commands; and for shape selections seven, eight, and nine, we use rounded rectangle commands.

Within each shape case, a CASE statement determines the particular command to be executed. For the first five commands in each case, we frame, paint, erase, invert, or fill the shape. In the sixth case, we erase the screen. There is no seventh case, since the seventh item of this menu is a disabled dashed line. However, there is an eighth line (the "Quit" command) that sets "done" equal to true, causing the main loop to terminate and thereby terminating the program.

Doing the Apple Menu

The next procedure, "DoAppleMenu", handles the options available under the Apple menu. It expects one parameter to specify the item selected from the Apple menu.

It has two local variables — an integer "refNum," and a string "name" — which are used with desk accessories.

The procedure begins by checking if item one, "About Menu", was selected. If so, it calls the Dialog Manager's "Alert" function to display a message on the screen in an alert box. Here are the resource definitions associated with the alert box:

```
Type ALRT
  ,1001
  100 70 200 440
  1001
  4444

Type DITL
  ,1001
  3

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 10 30 360
Menu, a demonstration program for menus

  StatText Disabled
  30 10 50 360
Christopher L. Morgan, 1985
```

If the item selected is not "About Menu", we call "GetItem" to get the name of the associated desk accessory, then "OpenDeskAcc" to activate that particular accessory.

Selecting the Menu Item

The procedure "SelectMenu" determines which menu is selected (if any). It also tracks the selection of an item and calls the appropriate action for the selected item.

The "SelectMenu" procedure has one parameter: a long integer "selection" to pass menu selection information generated by the Menu Manager.

It has two local variables, both integers: "theMenu" specifies the menu, and "theItem" selects the particular item within the menu.

We begin this procedure by extracting the menu identification number and the item number from the "selection" parameter. The upper 16 bits of "selection" (its "hi" part) contain the identification number of the selected menu. The lower bits (its "lo" part) contain the item number of the selected item within that menu. We store its "hi" part in "theMenu", its "lo" part in "theItem".

We now use a CASE statement to determine which menu is selected (if any; a zero indicates no selection). The various cases of "theMenu" are identified by menu identification numbers.

If the Apple menu is selected, we call "DoAppleMenu", described previously. If the Command menu is selected, we call "DrawShape". If any of the remaining menus are selected, we call the "ClickItem" procedure to update the variable that holds the choice for that menu and update the check mark in that menu.

After the CASE statement, we call "HiliteMenu", passing a zero to unhighlight all menus. If we don't do this, the title of the selected menu remains highlighted.

The Menu Manager's "HiliteMenu" routine expects one parameter, an integer that contains the menu identification number of the menu item to be highlighted. If the value passed in this parameter does not match the identification number of any menu in the Menu Manager's list of menus, then the routine unhighlights any highlighted menu item. Since no menu identification number is equal to zero, passing a zero to this routine unhighlights all menus.

The Main Program

The main program has much the same structure as main programs in previous example programs. It consists of an initialization section and a main REPEAT loop controlled by "done".

The initialization section calls the procedure "SetUpSys" to initialize QuickDraw, the various managers, and some of our global variables, and then calls the "SetUpMenus" procedure to initialize the menus.

We begin the main REPEAT loop by calling the Desk Manager's "SystemTask" routine to allow all open desk accessories to perform any required "background" actions. For example, the control panel has several parts that blink or change periodically.

In the main loop, we have the usual IF statement that surrounds the "GetNextEvent" function. Recall that this Event Manager function returns a value of true if we are to handle the event in our program. The "GetNextEvent" function returns the event record "theEvt" as its second variable, which we use to drive the CASE statements in the rest of our REPEAT loop.

The ".what" field of the event record drives the main CASE statement. The cases are "mouseDown", "keyDown", "updateEvt", and "activateEvt". The last two are lumped together.

For "mouseDown", we have a CASE statement driven by the result returned from "FindWindow". We have five cases: "inMenuBar", "inSysWindow", "inDrag", "inContent", and "inGrow". Again, the last two cases are lumped together.

In our program, only one window is under direct program control. However, each desk accessory has its own window. The various cases of "FindWindow" select from among these windows as well as the menu bar at the top of the screen.

For the "inMenuBar" case of "mouseDown", we call our "SelectMenu" procedure, passing to it the value of an expression directly involving the Menu Manager's "MenuSelect" routine.

The "MenuSelect" routine tracks the menu selection process, including pulling down the menus and highlighting their items as the mouse is held down and moved around. When the button is released, the routine returns (as a Pascal function) with long integer selection information containing the menu identification number and the item number. If no item is selected, it returns with a value of zero.

The "MenuSelect" routine expects a single parameter that is the location of the mouse in global screen coordinates at the time that the mouse button was pressed. In this case, we pass to it "theEvt.where" from the Event Manager. Interestingly, the entire menu selection process happens on a single line of Pascal during the evaluation of a single expression that is passed as a parameter.

For the "inSysWindow" case of "mouseDown", we call "SystemClick" to allow the currently selected desk accessory to perform a given action. This desk accessory is associated with the currently selected window.

For the "inDrag" case of "mouseDown", we call "DragWindow". As in other programs, this allows the user to drag the window around the screen but not off the screen and not into the menu bar.

For the "inContent" and "inGrow" cases of "mouseDown", we call "SelectWindow" to bring the selected window to the front and highlight it. When a desk accessory becomes active, its window comes to the front, deselecting all other windows including our display window. Having a way to "select" windows allows us to bring our own display window to the front again.

For the "keyDown" case, we also call "SelectMenu". This provides a way to select menu items by using the keyboard rather than the mouse.

In the previous case, we passed an expression that involved the mouse's "MenuSelect" routine. This time, we pass an expression involving the keyboard's "MenuKey" routine to "map" keys from the keyboard to those menu items that have key equivalents. In our program, these are the items in the Command menu. We find the ASCII code for the key in the lowest eight bits of the ".message" field of the event record. We then use "Chr" to convert it to type "Char" and "MenuKey" to map it to the long integer selection information.

For the "updateEvt" and "activateEvt" cases, we do an empty "BeginUpdate"/"EndUpdate" sequence. This allows the desk accessories to be properly updated as windows move around. Though our program doesn't "know" enough to explicity redraw a desk accessory when it needs updating, the "BeginUpdate" sequence signals the desk accessory to update itself at this point.

Before the "BeginUpdate", we point to the window to be updated by loading the ".message" field of "theEvt" into "theWindow", we use the type "windowPtr" like a function to coerce this quantity from type "LongInt" to type "windowPtr".

This takes care of all cases in our program, completing the REPEAT loop and the program.

Summary

In this chapter, we have studied the Menu Manager through an example program that manipulates seven menus, including the Apple menu filled with desk accessories and an information entry that activates an alert. The program also illustrates the various shapes, drawing actions, and drawing attributes available in QuickDraw, which menus organize in easy-to-use form.

The following ROM routines are covered in this chapter:

DL-InitDialogs
TE-TEInit
MN-CheckItem
MN-InitMenus
MN-NewMenu
MN-AddResMenu
MN-GetMenu
MN-InsertMenu
MN-DrawMenuBar
QD-PenPat
QD-PenSize
QD-PenMode
QD-FrameOval
QD-PaintOval
QD-EraseOval
QD-FillOval
QD-PaintRoundRect
QD-EraseRoundRect
QD-InvertRoundRect
QD-FillRoundRect
TU-HiWord
MN-GetItem
DS-OpenDeskAcc
MN-HiliteMenu
DS-SystemTask
MN-MenuSelect
DS-SystemClick
MN-MenuKey

10

Text and Files

This chapter covers the following new concepts:

- **The File Manager**
- **The Package Manager**
- **The Standard File Package**
- **Creating, Opening, and Closing Files**
- **Writing to and Reading from Files**
- **Text Edit**
- **Text Records**
- **Editing and Displaying Text**
- **Scrolling Text**

This chapter explores files and text. An example program illustrates how to program these two essential operations of applications programs.

Both text and files involve the management of blocks of information. Files store these blocks on disk and allow them to be transferred to and from memory. Text manages these blocks in memory. Of course, files can store information other than text: the way files are programmed is independent of the type of information they contain.

Our example program also uses many of the features — menus, windows, dialogs, and alerts — introduced in previous chapters. This program illustrates how a complete application should work, providing a fitting conclusion to the book. The challenge is to combine these old

concepts and handle the new concepts of text and files. As we see, everything fits into a grander structure controlled by a very simple but powerful main program that can run without change for a variety of applications and serves as an overview to Macintosh applications programming.

We could introduce text without files, but then our example would be useless: we would have no way to save the text or to conveniently supply examples of text to test our program.

This chapter discusses three new managers: the File Manager, Text Edit, and the Package Manager. The File Manager is considered part of the Operating System. It provides access to the file systems of the disks. It calls the lower levels of the system, such as the disk drivers. Text Edit is considered part of the Toolbox. It provides routines to edit blocks of text in memory. It calls lower levels, such as the Memory Manager. The Package Manager provides access to ROM-like routines that go beyond those in the ROM.

The Example Program

The example program is a simple text editor. Three menus define its major functions (see Figure 10-1).

The first menu, titled with the Apple symbol, allows the user to select desk accessories. The second menu, titled "File", has entries "New", "Open", "Close", "Save", and "Save As..." to set up, load, and save files, and "Quit" to exit the program. The third menu, titled "Edit", has entries "Cut", "Copy", and "Paste" to perform standard editing functions.

The first menu connects the program with other features and capabilities of the Macintosh. The second and third menus make this program into a simple text editor. Other menus could easily be added to provide options such as variable fonts, text faces, and text sizes.

The program also displays what is called the text window. At first, the text window is "Untitled" and empty. Later, it fills with text and is titled with the name of the file currently being edited. The text window has several features, including a vertical scroll bar that allows the user to scroll through the text, drag bars that allow the user to move the window, and a grow icon that allows the user to resize it.

Now let's examine each menu in more detail.

The Apple Menu

The Apple menu contains the desk accessories. We discussed these in Chapter 9.

The File Menu

The File menu contains seven entries: "New", "Open", "Close", "Save", "Save As...", a disabled entry filled with dashes, and "Quit" (see Figure 10-2).

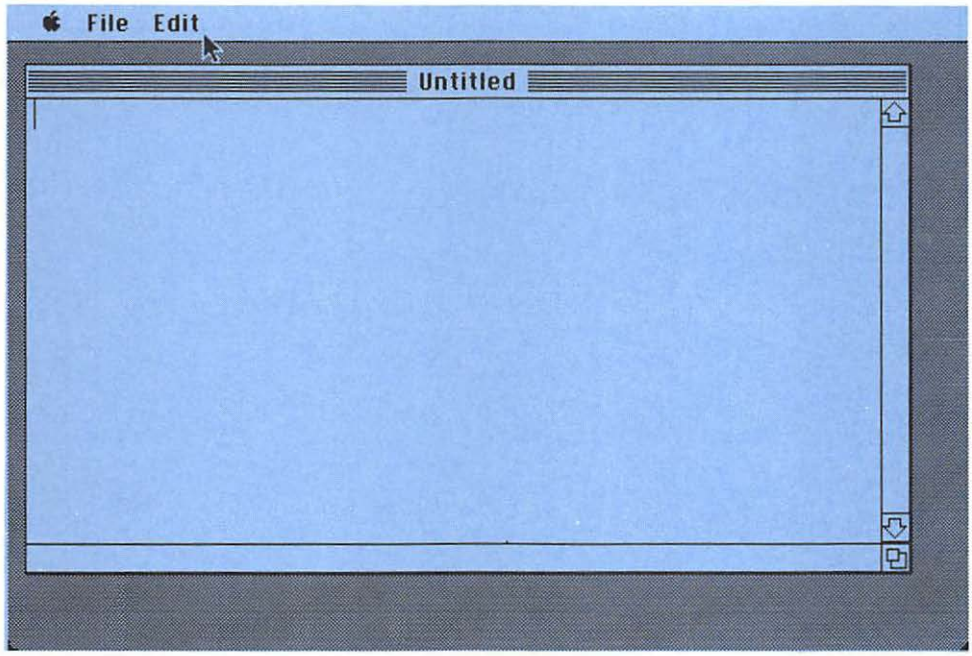
The entry "New" sets up an "untitled" text window. The entry "Open..." loads a selected text file, displaying its contents in the text window and its name in the title bar.

The three dots after the word "Open" indicate that a dialog appears when this item is selected (see Figure 10-3). This dialog is a standard dialog that displays the available text files. The user can select a file by double clicking it or by clicking an "open" button. The dialog also lets the user change disks.

The entry "Close" saves the file if it has been modified and makes the text window disappear. The entry "Save" of the File menu causes the current file to be saved on the disk.

The "Save" command proceeds without any special selection dialog. In contrast, the entry "Save As..." displays a standard dialog (see Figure 10-4) that allows the user to select the drive and file name under which

Figure 10-1. The Menus



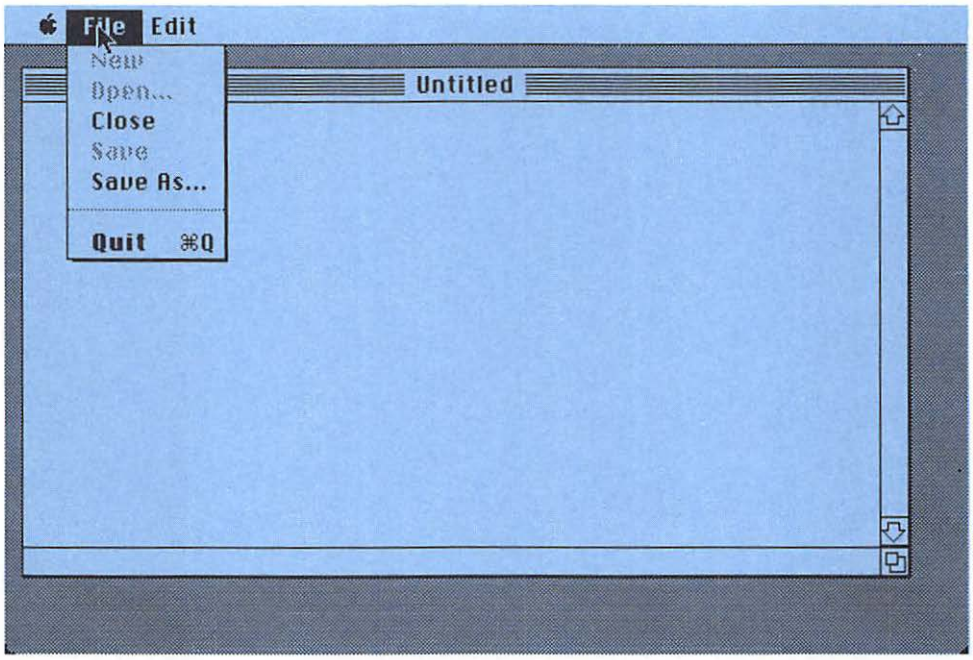
the file will be saved. Again, the three dots indicate that a dialog is to appear, requesting further information from the user. When we study this program, we see how these standard file dialogs, which open and save files, are available through the Package Manager.

To prevent undesirable actions, such as opening a text file when one is already open, the items in the file menu are selectively enabled and disabled. We explore this in detail when we study the workings of the program.

The Edit Menu

The Edit menu contains three entries: "Cut", "Copy", and "Paste" (see Figure 10-5). They preform their standard operations on our text in our text window. They also perform standard operations on desk accessories. However, the program does not allow transfer of text between our text window and any desk accessory. Such a transfer is handled by the Scrap Manager, which is not studied here.

Figure 10-2. The File Menu

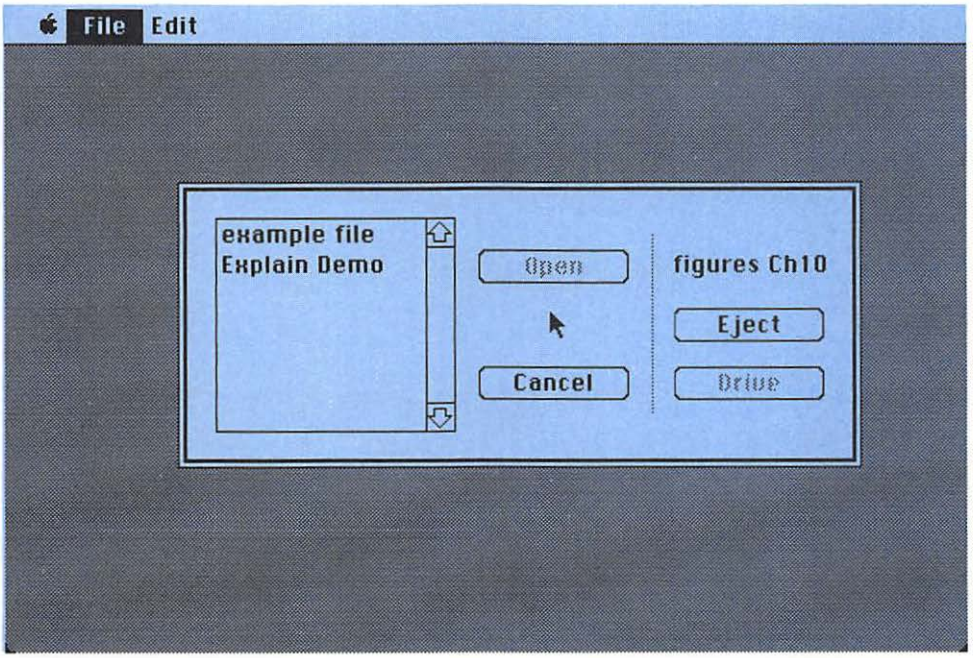


When not using the menus, the user can enter, view, and edit text in the text window. Notice that the window does not automatically scroll when it reaches the bottom of the screen. This feature is not built into the Macintosh; it must be programmed by the applications programmer.

Although this program is long, it contains many familiar routines and a modular structure that makes it easy to understand how the new parts fit in.

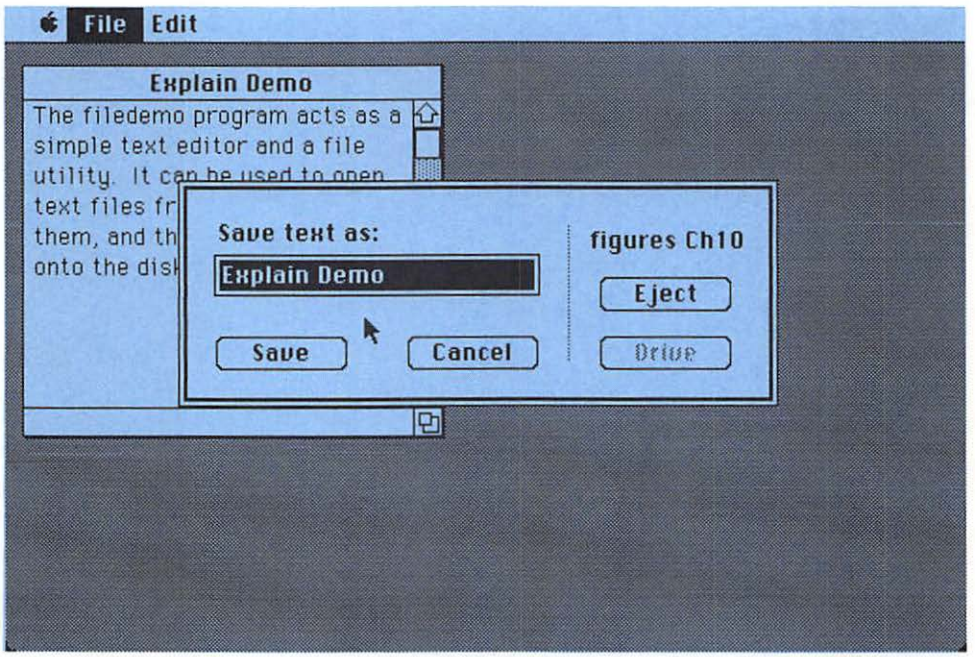
```
PROGRAM FileDemo;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf,
      {$U Obj/PackIntf    } PackIntf;
```

Figure 10-3. The Standard File Open Dialog



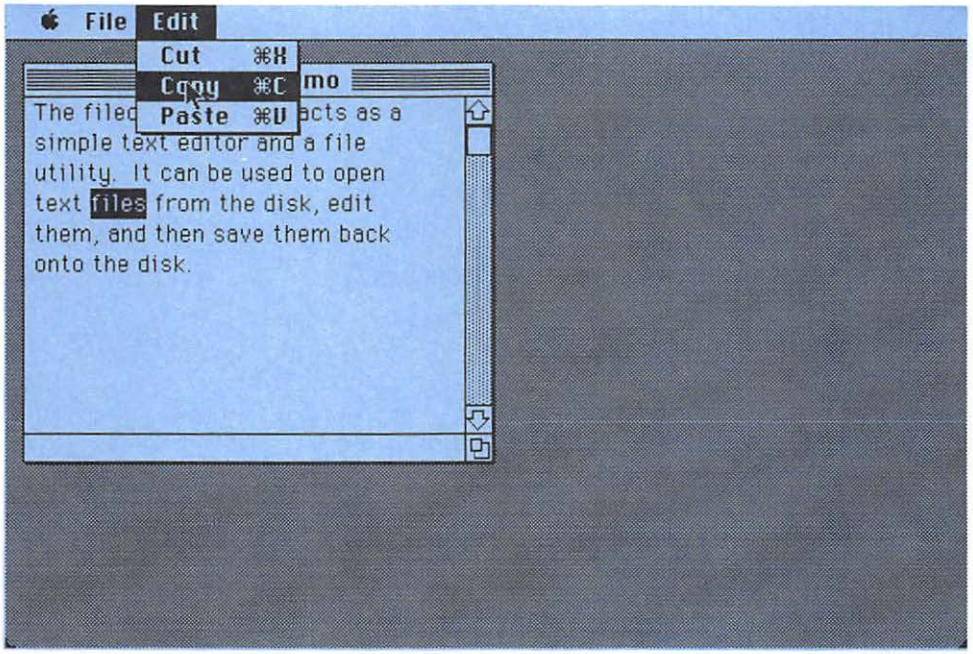
```
CONST
   {menu IDs}
   appleMenu = 1000;    { desk accessory menu}
   FileMenu  = 1001;    { File menu}
   EditMenu  = 1002;    { Edit menu}
   lastMenu  = 3;       { number of menus}

   {common dialog and alert items}
   OKBtn     = 1;
   cancelBtn = 2;

VAR
   done, present, titled, modified: BOOLEAN;
   dragBnds, sizeBnds, dRect, vRect: Rect;
   where: Point;
   myMenus: ARRAY [1..lastMenu] OF MenuHandle;
   theEvt: EventRecord;
   theWindow, textWindow: WindowPtr;
   vsbar: ControlHandle;
   theDialog: DialogPtr;
   fRefNum, vRefNum: INTEGER;
```

Figure 10-4. The Standard File Save As Dialog



```
    fName: Str255;
    theTE: TEHandle;

PROCEDURE SetLimits;
    BEGIN
        WITH screenBits.Bounds DO
            SetRect(dragBnds,left+4,top+24,right-4,bottom-4);
        SetRect(sizeBnds, 50, 50, 512, 342);
        SetPt(where,100,100);
    END;

PROCEDURE SetUpMenus;
    VAR
        I : INTEGER;
    BEGIN
        InitMenus;

        myMenus[1] := GetMenu(appleMenu);
        myMenus[2] := GetMenu(FileMenu);
        myMenus[3] := GetMenu(EditMenu);
```

Figure 10-5. The Edit Menu



```
      AddResMenu(myMenus[1], 'DRVR');
      FOR I:= 1 TO lastMenu DO InsertMenu(myMenus[I],0);
      DrawMenuBar;
   END; {of SetUpMenus}

PROCEDURE SetUpWindows;
   BEGIN
      textWindow := GetNewWindow(1000,NIL,POINTER(-1));
      vsbar:= GetNewControl(1000,textWindow);
   END;

PROCEDURE UpdateFState;

   PROCEDURE MenuItemEnable(theItem: INTEGER; enabled: BOOLEAN);
      BEGIN
         IF enabled THEN EnableItem( myMenus[2],theItem)
                    ELSE DisableItem(myMenus[2],theItem);
      END;

   BEGIN
      MenuItemEnable(1,NOT present);          {New}
      MenuItemEnable(2,NOT present);          {Open...}
      MenuItemEnable(3,present);              {Close}
      MenuItemEnable(4,titled AND modified);  {Save}
      MenuItemEnable(5,present);              {Save As...}
   END;

PROCEDURE UpdateScroll;
   VAR
      maxvalue: INTEGER;
   BEGIN
      maxvalue := theTE^^.nLines - 3;
      IF maxvalue<0 THEN maxvalue := 0;
      SetCtlMax(vsbar,maxvalue);
   END;

FUNCTION SetErrMess(theErr: OSErr): BOOLEAN;
   VAR
      ErrIndex, theItem: INTEGER;
      ErrMess, ErrStr: Str255;
      closeErr: BOOLEAN;
   BEGIN
      CASE theErr OF
         noErr:     ErrIndex := 2;
         bdNamErr:  ErrIndex := 4;
         fnfErr:    ErrIndex := 5;
         ioErr:     ErrIndex := 6;
         mFulErr:   ErrIndex := 7;
         nsvErr:    ErrIndex := 8;
```

```
            opWrErr:    ErrIndex := 9;
            tmfoErr:    ErrIndex := 10;
            eofErr:     ErrIndex := 11;
            paramErr:   ErrIndex := 12; {exact meaning depends}
            nsDrvErr:   ErrIndex := 13;
            dupFNErr:   ErrIndex := 14;
            dirFulErr:  ErrIndex := 15;
            vLckdErr:   ErrIndex := 16;
            wPrErr:     ErrIndex := 17;
            fnOpnErr:   ErrIndex := 18;
            rfNumErr:   ErrIndex := 19;
            dskFulErr:  ErrIndex := 20;
            fLckdErr:   ErrIndex := 21;
            wrPermErr:  ErrIndex := 22;
            posErr:     ErrIndex := 23;
            extFSErr:   ErrIndex := 24;
            Otherwise   ErrIndex := 3;
            END;
         GetIndStr(ErrMess,1000,ErrIndex);
         NumToStr(theErr,ErrStr);
         ParamText(ErrMess,ErrStr,'','');
         IF theErr <> noErr
            THEN theItem := StopAlert(1003,NIL);
         IF (theErr = opWrErr) OR (theErr = dskFulErr)
            THEN closeErr := SetErrMess(FSClose(fRefNum));
         SetErrMess := (theErr <> noErr);
      END;

   PROCEDURE NewTextWindow;
      BEGIN
         SetPort(textWindow);
         WITH textWindow^.portRect DO
            SetRect(dRect,left+4,top,right-19,bottom-15);
         vRect := dRect;
         theTE := TENew(dRect,vRect);
         SetCtlValue(vsbar,0);
      END;

   PROCEDURE NewFile;
      BEGIN
         NewTextWindow;
         GetIndStr(fName,1000,1);      {''Untitled''}
         SetWTitle(textWindow,fName);
         ShowWindow(textWindow);

         fName := '';
         vRefNum := 0;
```

```
      present  := TRUE;
      titled   := FALSE;
      modified := FALSE;
   END;

PROCEDURE OpenFile;
   VAR
      typeList:   SFTypeList;
      reply:      SFReply;
      CharCount:  LongInt;

   PROCEDURE FLCall(theErr: OSErr);
      BEGIN
         IF SetErrMess(theErr) THEN Exit(OpenFile);
      END;

   BEGIN
      typeList[0] := 'TEXT';
      SFGetFile(where,'',NIL,1,typeList,NIL,reply);
      IF reply.good THEN BEGIN
         NewTextWindow;

         FLCall(FSOpen(reply.fName,reply.vRefNum,fRefNum));
         FLCall(GetEOF(fRefNum,charCount));
         theTE^^.TELength := charCount;
         SetHandleSize(theTE^^.hText,charCount);
         FLCall(FSRead(fRefNum,charCount,theTE^^.hText^));
         FLCall(FSClose(fRefNum));

         TECalText(theTE);

         fName    := reply.fName;
         vRefNum := reply.vRefNum;
         SetWTitle(textWindow,fName);
         ShowWindow(textWindow);

         present  := TRUE;
         titled   := TRUE;
         modified := FALSE;
      END;
   END;

PROCEDURE SaveFile;
   VAR
      charCount: LongInt;

   PROCEDURE FLCall(theErr: OSErr);
      BEGIN
         IF SetErrMess(theErr) THEN Exit(SaveFile);
      END;
```

```
    BEGIN
        charCount := theTE^^.TELength;
        FLCall(FSOpen(fName,vRefNum,fRefNum));
        FLCall(FSWrite(fRefNum,charCount,theTE^^.hText^));
        FLCall(FSClose(fRefNum));
        modified := FALSE;
    END;

PROCEDURE SaveAsFile;
    VAR
        reply: SFReply;
        theErr: OSErr;
        charCount: LongInt;

    PROCEDURE FLCall(theErr: OSErr);
        BEGIN
            IF SetErrMess(theErr) THEN Exit(SaveAsFile);
        END;

    BEGIN
        SFPutFile(where,'Save text as:',fName,NIL,reply);
        IF reply.good THEN BEGIN
            charCount := theTE^^.TELength;
            theErr := FSOpen(reply.fName,reply.vRefNum,fRefNum);
            IF theErr = fnfErr THEN BEGIN
                FLCall(Create(reply.fName,reply.vRefNum,'    ','TEXT'));
                FLCall(FSOpen(reply.fName,reply.vRefNum,fRefNum));
                END
            ELSE FLCall(theErr);
            FLCall(FSWrite(fRefNum,charCount,theTE^^.hText^));
            FLCall(FSClose(fRefNum));

            fName    := reply.fName;
            vRefNum  := reply.vRefNum;
            SetWTitle(textWindow,fName);
            titled   := TRUE;
            modified := FALSE;
        END;
    END;

PROCEDURE CloseFile;
    BEGIN
        IF modified THEN
            CASE CautionAlert(1004,NIL) OF
                OKBtn:     IF titled THEN SaveFile
                                     ELSE SaveAsFile;
                cancelBtn: Exit(CloseFile);
                END;
```

```
      HideWindow(textWindow);
      TEDispose(theTE);
      present  := FALSE;
      titled   := FALSE;
   END;

PROCEDURE QuitFile;
   BEGIN
      IF present THEN CloseFile;
      done := NOT present;
   END;

PROCEDURE SetUpSys;
   BEGIN
      InitGraf(@thePort);
      InitFonts;
      InitWindows;
      TEInit;
      InitDialogs(NIL);
      SetEventMask(everyEvent);
      FlushEvents(everyEvent,0);

      SetLimits;
      SetUpWindows;
      SetUpMenus;
      NewFile;
      InitCursor;
      done := FALSE;
   END;

PROCEDURE UpdateSys;
   BEGIN
      SystemTask;
      UpdateFState;
      IF present THEN BEGIN
         TEIdle(theTE);
         UpdateScroll;
         END;
   END;

PROCEDURE DoAppleMenu(theItem: INTEGER);
   VAR
      refNum: INTEGER;
      name:   Str255;
   BEGIN
      If theItem = 1
         THEN theItem := Alert(1001,NIL)
         ELSE
```

```
            BEGIN
               GetItem(myMenus[1],theItem,name);
               refNum := OpenDeskAcc(name);
            END;
   END;

PROCEDURE DoFileMenu(theItem: INTEGER);
   BEGIN
      CASE theItem OF
         1: NewFile;
         2: OpenFile;
         3: CloseFile;
         4: SaveFile;
         5: SaveAsFile;
         7: QuitFile;
      END;
   END;

PROCEDURE DoEditMenu(theItem: INTEGER);
   BEGIN
      IF NOT SystemEdit(theItem+1) THEN BEGIN
         SetPort(textWindow);
         modified := (theItem in [1,3]);
         CASE theItem OF
            1: TECut(theTE);
            2: TECopy(theTE);
            3: TEPaste(theTE);
            END;
         END;
   END;

PROCEDURE SelectMenu(selection : LongInt);
   BEGIN
      CASE HiWord(selection) OF
         appleMenu:   DoAppleMenu(LoWord(selection));
         FileMenu:    DoFileMenu (LoWord(selection));
         EditMenu:    DoEditMenu (LoWord(selection));
         END;
      HiliteMenu(0); {to unhighlight selected menu in menu bar}
   END;

PROCEDURE WindowDrag(thePt: Point);
   BEGIN
      DragWindow(theWindow,thePt,dragBnds);
   END; {SelectMenu}

PROCEDURE ScrAction(theCtl: ControlHandle; partCode: INTEGER);
   VAR
      pageSize, delta: INTEGER;
      S, dS : Point;
```

```
    BEGIN
       WITH theTE^^ DO
          pagesize := (viewRect.bottom - viewRect.top) DIV lineHeight;
       CASE partCode OF
          inUpButton:   delta := -1;
          inDownButton: delta := +1;
          inPageUp:     delta := -pagesize;
          inPageDown:   delta := +pagesize;
          otherwise     Exit(ScrAction);
          END;

       SetPt(S,0,GetCtlValue(theCtl));
       SetCtlValue(theCtl,GetCtlValue(theCtl)+delta);
       SetPt(dS,0,S.v-GetCtlValue(theCtl));
       TEScroll(0,dS.v*theTE^^.lineHeight,theTE);
    END;

PROCEDURE WindowControl(thePt: Point);
    VAR
       theCtl  : ControlHandle;
       S, dS : Point;
    BEGIN
       IF (theWindow = frontWindow) AND present THEN BEGIN
          SetPort(theWindow);
          GlobalToLocal(thePt);
          IF PtInRect(thePt,theTE^^.viewRect)
             THEN TEClick(thePt,BitTst(@theEvt.modifiers,6),theTE)
             ELSE CASE FindControl(thePt,theWindow,theCtl) OF
                inUpButton, inDownButton, inPageUp, inPageDown:
                   IF TrackControl(theCtl,thePt,@ScrAction)<>0 THEN;
                inThumb:
                   BEGIN
                   SetPt(S,0,GetCtlValue(theCtl));
                   IF TrackControl(theCtl,thePt,NIL)<>0 THEN BEGIN
                      SetPt(dS,0,S.v-GetCtlValue(theCtl));
                      TEScroll(0,dS.v*theTE^^.lineHeight,theTE);
                      END;
                   END;
                END;
          END
       ELSE BEGIN
          SelectWindow(theWindow);
          DrawControls(theWindow);
          DrawGrowIcon(theWindow);
       END;
    END; {WindowControl}
```

```
PROCEDURE WindowGrow(thePt: Point);
   VAR
      Wsize : LONGINT;
      S : Point;
   BEGIN
      WSize := GrowWindow(theWindow, thePt, sizeBnds);
      IF WSize = 0 THEN Exit(WindowGrow);

      SetPort(theWindow);
      SetPt(S, loWord(WSize), hiWord(WSize));
      SizeWindow(theWindow, S.h, S.v, true);
      ClipRect(thePort^.portRect);

      SizeControl(vsbar,    16, S.v-13);
      MoveControl(vsbar, S.h-15,     -1);
      InvalRect(theWindow^.portRect);

      DrawGrowIcon(theWindow);
      IF present THEN
         WITH theTE^^ DO BEGIN
            viewRect.right  := viewRect.left+S.h-23;
            viewRect.bottom := viewRect.top +S.v-15;
         END;
   END; {WindowGrow}

PROCEDURE KeyEvent(theKey: Char);
   BEGIN
      IF BitTst(@theEvt.modifiers, 7) {check for command key}
         THEN SelectMenu(MenuKey(theKey))
         ELSE IF (textWindow = frontWindow) AND present THEN
            BEGIN
               TEKey(theKey, theTE);
               modified := TRUE;
            END;
   END; {KeyEvent}

PROCEDURE WindowUpdate;
   BEGIN
      theWindow := windowPtr(theEvt.message);
      SetPort(theWindow);
      IF theWindow = FrontWindow THEN ShowControl(vsbar)
                                 ELSE HideControl(vsbar);
      BeginUpdate(theWindow);
         EraseRect(theWindow^.portRect);
         IF (theWindow = textWindow) AND present THEN
            TEUpdate(theWindow^.visRgn^^.rgnBBox, theTE);
         DrawControls(theWindow);
         DrawGrowIcon(theWindow);
      EndUpdate(theWindow);
   END; {Update}
```

```
PROCEDURE WindowActivate;
   BEGIN
      WindowUpdate;
      IF present THEN
         IF ODD(theEvt.modifiers) THEN TEActivate(theTE)
                                  ELSE TEDeactivate(theTE);
   END; {Activate}

BEGIN  {main program}
   SetUpSys;
   REPEAT
      UpdateSys;
      IF GetNextEvent(everyEvent,theEvt) THEN
         CASE theEvt.what OF
            mouseDown:
               CASE FindWindow(theEvt.where,theWindow) OF
                  inMenuBar:   SelectMenu(MenuSelect(theEvt.where));
                  inSysWindow: SystemClick(theEvt,theWindow);
                  inDrag:      WindowDrag(theEvt.where);
                  inContent:   WindowControl(theEvt.where);
                  inGrow:      WindowGrow(theEvt.where);
                  END;
             keyDown, autoKey: KeyEvent(Chr(theEvt.message MOD 256));
             updateEvt:        WindowUpdate;
             activateEvt:      WindowActivate;
             END; {of what event}
   UNTIL done;
END.
```

External Files

The program begins with a USES section, which, in addition to standard files, also uses a file called "PackIntf". This allows us to use special file dialogs and conversion routines between numbers and strings.

Global Constants

The CONST section is a useful interface between the program and its resource file. It assigns names to various numbers that identify the menus and dialog items in the resource file. This allows us to develop the resource definition file without constantly changing numbers in the program.

The three menus are "appleMenu", "FileMenu", and "EditMenu". The "lastMenu" is set equal to three to indicate a total of three menus.

Two button items are used in an alert: "OKBtn" and "CancelBtn".

Global Variables

The VAR section declares a number of variables used globally in this program.

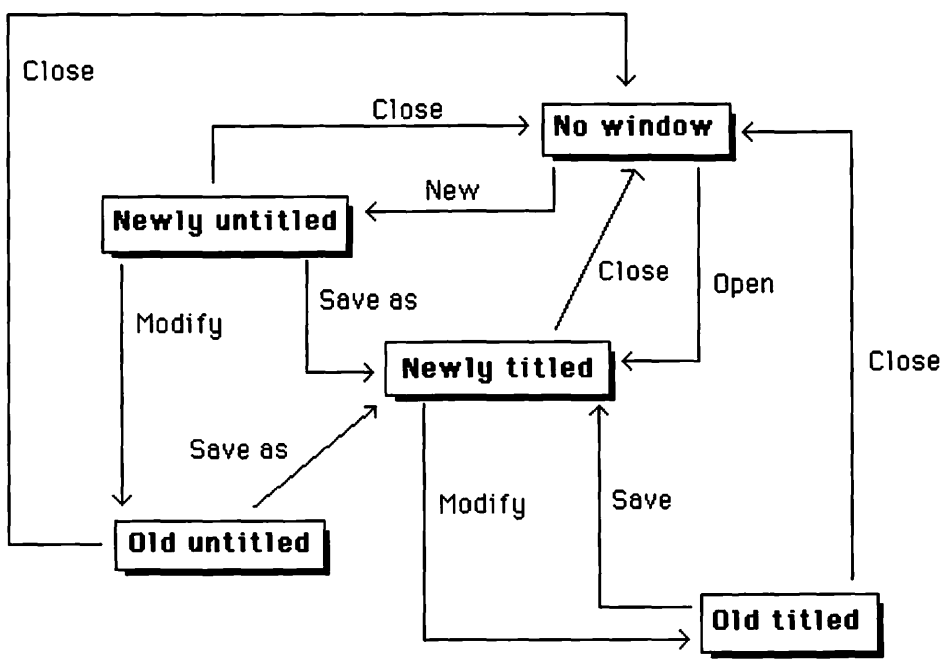
Four Boolean variables keep track of the program. The Boolean variable "done" controls the main loop to determine if the program is finished. The user controls "done" by the "Quit" command of the File menu. The three remaining Boolean variables set the file state in reference to other file commands.

File States

This program has five possible states for loading and saving files (see Figure 10-6). These states form a "finite state machine" whose state transitions are given by the file commands listed in the file menu.

The three global Boolean variables "present", "titled", and "modified" can form eight possible value choices. Here we allow only five of those possibilities:

Figure 10-6. File States



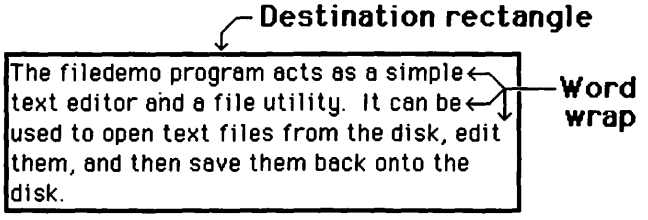
1. No window. In this state, "present" is FALSE, "titled" is FALSE, and "modified" is FALSE. This state occurs after a file closes.
2. Newly untitled. In this state, "present" is TRUE, "titled" is FALSE, and "modified" is FALSE. This state occurs after the text window is newly initialized, such as when the program starts or after the "New" command is issued.
3. Old untitled. In this state, "present" is TRUE, "titled" is FALSE, and "modified" is TRUE. This state occurs after text enters an untitled window.
4. Newly titled. In this state, "present" is TRUE, "titled" is TRUE, and "modified" is FALSE. This state occurs after a file is opened or saved but no changes have been made.
5. Old titled. In this state, "present" is TRUE, "titled" is TRUE, and "modified" is TRUE. This state occurs after a file is opened or saved and changes are made.

For each state, certain items from the file menu are enabled, others disabled. For example, in the "no window" state, the "New" and "Open" commands are enabled, but "Close", "Save", and "Save As" are disabled. We later see how.

The next four variables are rectangles that determine limits. The first rectangle, "dragBnds", provides the limits for dragging windows. The second rectangle, "sizeBnds", provides the limits for resizing windows.

The third and fourth rectangles size the text. The third rectangle, "dRect", is called the destination rectangle. The text is mapped into this rectangle using "word wrap". That is, the text is laid out so that it falls within the horizontal dimensions of this rectangle, breaking text lines only at word boundaries. In the vertical direction, text may continue beyond the bottom of this rectangle. This rectangle may be larger or smaller than the screen (see Figure 10-7). It defines the "full" image of the text.

Figure 10-7. The Destination Rectangle for Text



The fourth rectangle, "vRect", is the viewing rectangle. It acts like a "window" for viewing the text. Only that part of the text that falls within the view rectangle (and the visible parts of the window) is displayed (see Figure 10-8). To scroll, move the destination rectangle while keeping the view rectangle fixed.

The next global variable is a point, "where", that locates the upper left corner of the alerts in this program.

Next, two window pointers are declared: "theWindow" is a general window pointer, and "textWindow" points to our text window. The global variable "vsbar" is a control handle to the vertical scroll bar of the text window.

Next, "myMenus" is an array of menu handles to access our four menus. An event record, "theEvt", tracks events as in previous programs.

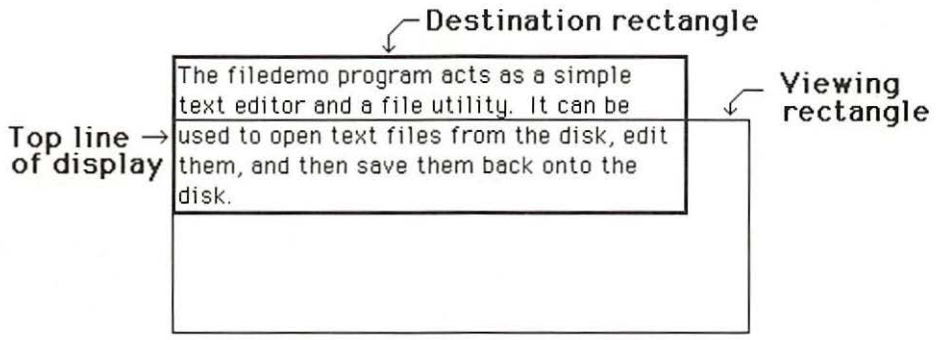
A dialog pointer, "theDialog", points to the various dialogs and alerts in the program. It is reused a number of times, since dialogs are disposed of once they are closed.

Two global integers, "fRefNum" and "vRefNum", hold reference numbers for files and volumes, respectively. The string "fName" holds the current file name.

The text handle "theTE" is a handle to the text edit record, described by the following Pascal declarations:

```
TEPtr      = ^TERec;
TEHandle   = ^TEPtr;
TERec      = RECORD
                 destRect:     Rect;
                 viewRect:     Rect;
                 selRect:      Rect;
```

Figure 10-8. The Viewing Rectangle for Text



```
    lineHeight:   INTEGER;
    fontAscent:   INTEGER;
    selPoint:     Point;
    selStart:     INTEGER;
    selEnd:       INTEGER;
    active:       BOOLEAN;
    wordBreak:    LONGINT;
    clikLoop:     LONGINT;
    clikTime:     LONGINT;
    clikLoc:      INTEGER;
    caretTime:    INTEGER;
    caretState:   INTEGER;
    just:         INTEGER;
    TELength:     INTEGER;
    hText:        Handle;
    recalBack:    INTEGER;
    recalLines:   INTEGER;
    clikStuff:    INTEGER;
    crOnly:       INTEGER;
    txFont:       INTEGER;
    txFace:       INTEGER;
    txMode:       INTEGER;
    txSize:       INTEGER;
    inPort:       GrafPtr;
    highHook:     PTR;
    caretHook:    PTR;
    nLines:       INTEGER;
    LineStarts:   ARRAY [0..16000] OF INTEGER;
END;
```

A text handle (type "TEHandle") points to a text pointer (type "TEPtr") that points to a text record (type "TERec"). A text record has a number of fields to describe the appearance of the text and its internal structure.

The first two fields, ".destRect" and ".viewRect", contain the current values of the destination and view rectangles, respectively (see Figure 10-9). The third field, ".selRect", contains a rectangle to delimit the selection area in the text's grafPort.

The fourth field, ".lineHeight", contains the line height or vertical distance between lines (see Figure 10-10).

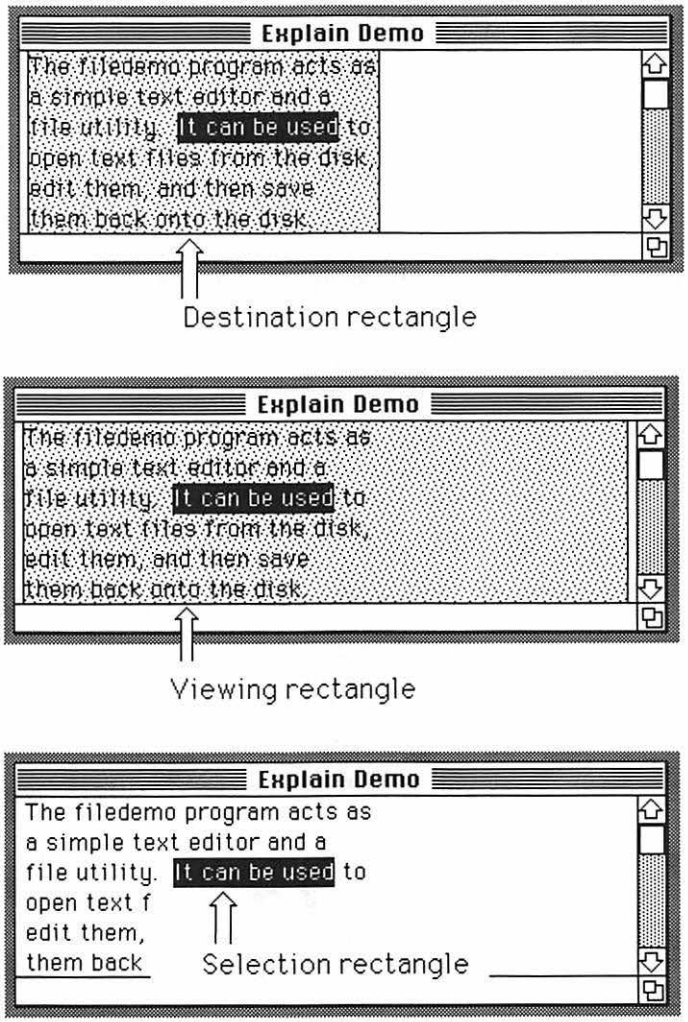
The ".fontAscent" field contains the font ascent of the text (see Figure 10-11).

The ".selPoint" field is the location of the mouse when clicked within the text display on the screen.

The ".selStart" and ".selEnd" fields give the starting and editing limits of the selection range. If these agree, then an insertion point occurs at their common value. This insertion point is indicated by a caret (blink-

ing vertical line). These quantities range from zero (before the first character) to the number that represents the character position after the last character of text. However, since these and the ".TElength" field are integers, an edit record cannot control more characters than the largest 16-bit signed integer, which is 32,767. Documents with more characters should be divided into smaller edit records, perhaps by paragraph or page.

Figure 10-9. Destination, View and Selection Rectangles



The ".active" field indicates if the text is active (caret is blinking and editing working). A nonzero value indicates active.

The ".wordBreak" field contains a long integer that points to the procedure for handling word breaks. The ".clikLoop" field contains a long integer that points to the procedure for handling mouse clicks.

The ".clikTime" field contains a long integer that specifies the time of the first click of the mouse button. This is useful for handling double clicks. The ".clikLoc" field contains an integer that specifies the character position of the mouse where clicked.

The ".caretTime" field contains a long integer that specifies the time for the next blink of the caret. The ".caretState" contains an integer whose bits act like Boolean variables that specify the active and on/off state of the caret.

The ".just" field determines the justification of the text. A value of zero indicates left justification, a value of one indicates centered text, and a value of minus 1 indicates right justification.

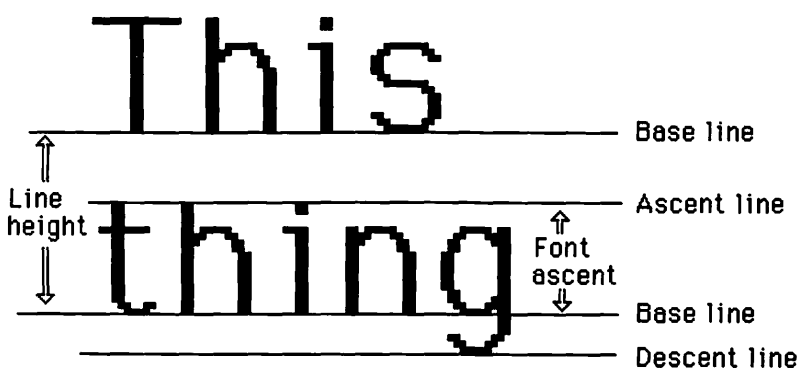
The ".TElength" field contains the length of the text; that is, the number of characters in the text. As noted previously, this is a 16-bit integer, thus limiting the number of characters controlled by an edit record.

The ".hText" field is a handle to the text itself.

The ".recalBack", ".recalLines", and "clikStuff" fields are integers that the Text Manager uses to recalculate "line starts" for the text. These places within the text occur at the beginning of lines when the text is displayed on the screen (see Figure 10-11).

The ".crOnly" turns wrapping text line on and off (see Figure 10-12). If this field is zero, then standard word wrap is used. If this field is minus

Figure 10-10. Line Height and Font Ascent



one, then lines that extend beyond the horizontal limits of the destination rectangle are truncated (chopped off). In that case, only the carriage return terminates a displayed line of text before it is truncated.

The next four fields — ".txFont", ".txFace", ".txMode", and ".txSize" — are attributes of the text. Normally, an application has menus that allow the user to change these fields.

The ".inPort" field points to a grafPort associated with the text. This is the grafPort that was current when the text opened.

The ".highHook" and ".caretHook" fields contain pointers to routines for advanced programmers to customize the highlighting of carets and selection areas.

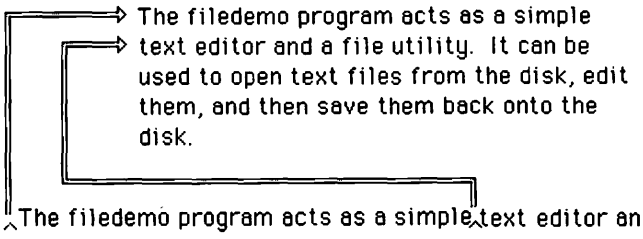
The ".nLines" field contains an integer that specifies the number of lines of text controlled by the text record. The ".lineStarts" field is an array of integers that specify where each line starts.

Functions and Procedures

This program combines a number of features introduced in Chapter 9, so many of the functions and procedures in this program should be familiar. However, this program contains new routines and new wrinkles to old routines.

The procedures are ordered according to "level"; that is, they are arranged according the usual Pascal calling sequence wherein each routine can be called only after it is defined (no forward references). The procedures and functions are also grouped according to their function. For example, all lowest level initialization procedures are grouped together.

Figure 10-11. Line Starts



Low-Level Initialization Procedures

The first few procedures — "SetLimits", "SetUpWindows", and "SetUpMenus" — initialize subsystems of the Macintosh, including rectangle and point limits, windows, and menus. These routines reside at the lowest level of our program; that is, they don't call other procedures. You can place your own low-level initialization routines in this section for any program you write for the Macintosh.

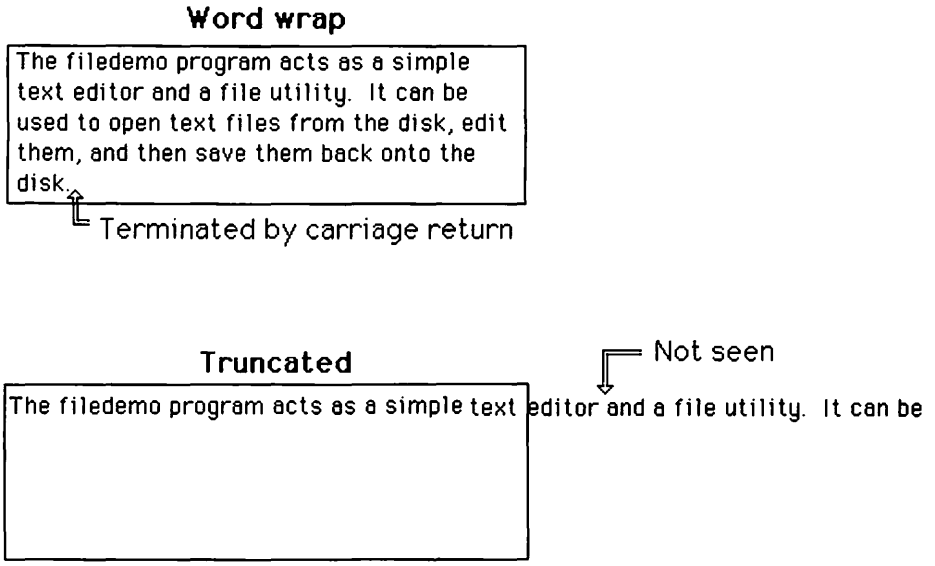
Setting Limits

The procedure "SetLimits" sets the limit values for several rectangles and a point. The rectangles are "dragBnds" and "sizeBnds"; the point is "where". The two rectangles are set with the standard limits. The point "where" locates the upper left corner of alert boxes and is set equal to (100,100), placing the alerts toward the middle of the screen. Your program may have other variables to be initialized. This is a good place to do it.

Setting Up Menus

The procedure "SetUpMenus" initializes the menus. It has one local variable, "I", an index to a loop.

Figure 10-12. Wrapping and Truncating Text Lines



It calls "InitMenus" to initialize the Menu Manager. It calls "GetMenu" to get the definition of the four menus from the resource file, initialize the menus, then return a handle to them. We store these handles in the array "myMenus".

These menus are defined in the resource definition file as follows:

```
Type MENU
  ,1000
  \14
       About FileDemo...
       (--------------

  ,1001
  File
     New
     Open...
     Close
     Save
     Save As...
     (---------
     Quit/Q <B

  ,1002
  Edit
     Cut/X
     Copy/C
     Paste/V
```

In this program, as in the last, the Apple menu is partly defined in the resource file and partly defined by using "AddResMenu" to add all available desk accessories to it. You may want to add other system resources to other menus at this point.

The "SetUpMenus" procedure concludes with a FOR loop that uses "InsertMenu" to add each menu to the master list of menus, then calls "DrawMenuBar" to draw the menu titles on the screen.

Setting up Windows

The procedure "SetWindows" initializes the text window and its scroll bar. It calls "GetNewWindow" to initialize the text window and obtain a handle to it. It calls "GetNewControl" to initialize the scroll control, attach the scroll control to the text window, and obtain its handle. These are defined in the resource definition file as:

```
Type WIND
     ,1000
     Untitled
     50 10 300 480
     Visible NoGoAway
     0
     0


Type CNTL
  ,1000
  vertical scroll bar
  -1 455 235 471
  Invisible
  16
  2
  0 0 450
```

Other windows and controls in your program should be initialized here.

Low-Level Updating Routines

The next two procedures, "UpdateFState" and "UpdateScroll", update various variables and subsystems as part of the main event loop. They are also low level — they do not call other procedures or functions in our program.

The procedures are called by the procedure "UpdateSys", which is called each time the main event loop is executed. Other routines are also called in this master update procedure, but they are already defined in external files. We see how such updating routines fit together when we study this master "UpdateSys" procedure.

You may wish to add other low-level routines at this place in your program.

Updating the File State

The procedure "UpdateFState" updates the file menu according to the state of file loading and saving.

As discussed, the file state can be described by the three Boolean variables "present", "titled", and "modified". Only certain commands should be enabled for any particular state.

We use the Menu Manager routines "EnableItem" and "DisableItem" to enable and disable menu items in the file menu. These routines expect two parameters. The first is a handle to the particular menu; the second is an integer index to the particular item.

A subprocedure, "MenuItemEnable", does the actual enabling and disabling. It expects two parameters: the item's number and a Boolean that specifies if the item should be enabled.

The first two items, "New" and "Open...", are enabled only if the text window is *not* "present". The items "Close" and "Save As..." are enabled only if the text window is "present".

The "Save" menu item is handled differently. It is enabled when the text window is "titled" *and* "modified", but disabled otherwise.

Updating Scroll Limits

The procedure "UpdateScroll" updates the scroll limits and highlighting according to the amount of text in the system. It has one local variable, "maxvalue", an integer.

The procedure calls "SetCtlMax" to set the maximum value according to the formula:

```
max(0,theTE^^.nLines - 3)
```

Since this Pascal doesn't have a maximum value function, we first use an inequality to ensure that we don't set the maximum value less than zero. The local variable "maxvalue" holds the value during computation.

File Menu Procedures and Functions

The next procedures and functions in this program illustrate how to program the loading and saving of files. They implement the file commands in the File menu. To support these commands, we first need an error handler function and a routine to set up a new text window.

Errors

The function "SetErrMess" is a general-purpose routine to handle errors. It has one parameter, of type "OSErr" (equal to type INTEGER). This parameter contains the error code returned from a File Manager routine.

Our "SetErrMess" function returns a Boolean variable that is true only if an error occurs.

The function has five local variables. The first two are integers: "ErrIndex" indexes into our list of errors in our resource definition file, and "theItem" is used in conjunction with a special error alert. The next two are strings: "ErrMess" holds the error message, and "ErrStr" holds the error number. Both strings are displayed in the stop alert. The last local variable is the Boolean variable "closeErr", which holds a result returned from a close file command that is issued in response to certain errors.

The "SetErrMess" function begins by using a CASE statement to map the error code "theErr" to the index "ErrIndex" in our list of errors.

We then call "GetIndStr" to look up the appropriate string in our resource file. The "GetIndStr" routine expects three parameters: a string that is returned, the resource number of a string list resource (type 'STR#'), and an index into this list. This list is defined in the resource definition file as:

```
Type STR#
  ,1000
Untitled
No error
Unknown Error
Bad file or volume Name
File not found
Disk I/O Error
Memory full
No such volume
File already open for writing
Too many files open
End of file
Bad number
No such drive
Duplicate file name
Directory full
Software volume lock
Hardware volume lock
File not open
Bad reference number
```

```
Disk full
Permission denied to access file
File position out of range
External file system
```

Next, we call "NumToStr" to convert the error code to the string "ErrStr". We use "ParamText" to load these strings for display in the next dialog or alert box.

If an error (theErr <> noErr) occurs, we call "StopAlert" to display the appropriate alert. This alert is defined in the resource file as follows:

```
Type ALRT
  ,1003
  100 70 200 440
  1003
  7654

Type DITL
  ,1003
  3

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 150 50 360
File Error: ^0

  StatText Disabled
  60 150 90 360
ID number: ^1
```

If the error was "File open for writing" (opWrErr) or "Disk full" (dskFulErr), we attempt to close the file, calling the "SetErrMess" recursively and passing the File Manager's "FSClose" routine as its parameter.

The last statement in this function returns the Boolean value of the function determined by whether or not an error occurred.

Making a New Text Window

The procedure "NewTextWindow" sets up a new text window for the "New" or "Open..." commands.

It begins by calling "SetPort" to set the current grafPort to the textWindow grafPort. Next, it sets the global destination rectangle "dRect" and view rectangle "vRect" to just within the port rectangle, making room

for scroll bars and a small margin on either side of the "page". These margins keep the text looking tidy, as if it were on a sheet of paper.

In particular, the left-hand limits are four pixels to the right of the left-hand limit of the port rectangle, the right-hand limits are nineteen pixels to the left of the right-hand limit of the port rectangle, the upper limits agree, and the bottom limits are fifteen pixels above the lower limit of the port rectangle.

Next, it calls "TENew" to open up the text record with the destination and view rectangles just set. The "TENew" routine returns a handle to the text record. We store this handle in "theTE".

Finally, we call "SetCtlValue" to set the control value for the scroll bar equal to zero.

New File Command

The "NewFile" procedure implements the "New" command of the File menu. The "NewFile" procedure begins by calling our "NewTextWindow" procedure to set up a new text window. It then calls "SetWTitle" to set the title of this window to "Untitled", and then uses "ShowWindow" to bring the text window into view.

It sets the file name stored in the global "FName" equal to the empty string and the volume reference number stored in the global variable "vRefNum" equal to zero (the default drive).

Finally, we set the Boolean variables "present" equal to true, "titled" equal to false, and "modified" equal to false. This puts us into the "newly untitled" state.

Notice that this routine does not call any disk commands.

Open File Command

The "OpenFile" procedure implements the "Open..." command of the File menu. It has three local variables: "typeList" is of type "SFTypeList" and specifies the desired file type, "charCount" is a long integer that temporarily stores the length of the file, and "reply" is of type "SFReply" and holds the results from the Package Manager's standard file selection routine.

The type "SFTypeList" is defined by the following Pascal declaration:

```
SFTypeList = ARRAY [0..3] OF OSType;
```

This is simply an array of four items, each a four-character string specifying a file type.

The type "SFReply" is defined by the following Pascal declaration:

```
SFReply = RECORD
            good:      BOOLEAN;
            copy:      BOOLEAN;
            FType:     OSType;
            vRefNum:   INTEGER;
            version:   INTEGER;
            fName:     STRING[63];
          END;
```

This is a record structure. The first field, ".good", is a Boolean variable that specifies if a file selection is to be made (OK button or Return key is hit). The second field, ".copy", is not used. The third field, ".FType", is the file type of the selected file. The fourth field, ".vRefNum", is an integer that specifies the particular disk to find the file on. The fifth field, ".version", is the version number of the Operating System associated with the disk. This should be zero for now. The sixth field, ".fName", is the file name.

The "OpenFile" procedure has one subprocedure, "FLCall", that helps call disk routines. This subprocedure has one parameter, of type "OSErr". If this parameter is not equal to "noErr", it displays an alert that shows the error message and its identification number and causes a quick exit from our "OpenFile" routine.

The "FLCall" subprocedure has only one Pascal statement, an IF..THEN statement that calls our "SetErrMess" routine in the IF part to determine if an error occurred. The THEN part is an "Exit" from the "OpenFile".

The "OpenFile" procedure begins by setting the zeroth entry of "typeList" equal to "TEXT", which is the type of file we wish to find. It then calls the Standard File Package's "SFGetFile" routine to select the file to be opened. It displays a standard dialog that allows the user to scroll through all files of the specified type(s) (see Figure 10-14).

The "SFGetFile" routine is not part of the ROM. It is part of the software called the *Standard File Package*, stored as a resource in the Operating System file called "System" (see Figure 10-13). In our program, we simply call the routine "SFGetFile". However, the Macintosh performs a rather complicated sequence of events in response to this call. The beginning of the "SFGetFile" routine is in the external UNIT "PackIntf". This routine calls one of the Package Manager ROM routines ("Pack3") to execute this resource. The "Pack3" routine gives access to all routines in the *Standard File Package*. A special function *selector* number (in this case, two) must be passed to "Pack3" to determine which routine in the package is executed.

In addition to the Standard File Package, the following packages are available: the International Utilities Package, the Disk Initialization Package, the Floating Point Package, the Transcendental Functions Package, and the Binary-Decimal Conversion Package. Each package is called by a different "Pack" routine.

The "SFGetFile" routine expects seven parameters. The first parameter is a point that determines where the upper left corner of the dialog window appears. In our program, we pass the global point "where", which we set in our "SetLimits" procedure.

The second parameter, a string, is ignored. In earlier versions of the Macintosh, it served as a prompt in the dialog, but it is no longer used. It is retained for compatibility. In our program, we pass the string "Open text file:".

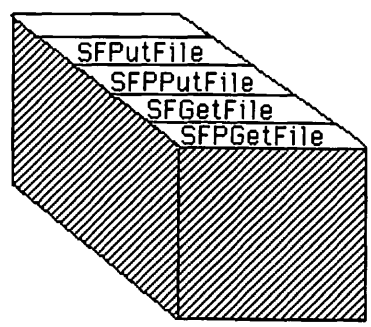
The third parameter is a procedure pointer to a filter procedure to help determine which files are displayed in the dialog for the file selection process. In our program, we pass NIL to indicate no filter.

The fourth parameter is an integer that specifies how many types we are looking for. If this parameter is set equal to minus one, then it looks for all types in our dialog. In our program, we pass a value of one to indicate just one type of file. The fifth parameter is a file type list. In our program, we pass the type list just initialized.

The sixth parameter is a procedure pointer to a filter procedure for dialog. In our program, we pass NIL to indicate no special filter. The seventh parameter is a reply record of type "SFReply", discussed previously. In our program, we place "reply" here to receive our responses from this dialog.

If the ".good" field of "reply" is true, indicating that a valid file selection was made in the dialog, we attempt to get the text from the file.

Figure 10-13. The Standard File Package



We start by calling our "NewTextWindow" to set up a new text window. Then we call a series of File Manager routines to open, read from, then close the selected file. In each case, we call our "FLCall" routine, passing it the expression consisting of the particular file routine. This causes the error code generated by that file routine to be passed to the "FLCall" routine, which in turn passes it to our "SetErrMess" procedure. If there is no error, the file routine is executed without further ado.

Let's look at each file routine in detail. The routine "FSOpen" attempts to open a file. If it is successful, it makes the file "active", setting up the file so that it can be read or written to. The "FSOpen" routine expects three parameters: a file name, a volume reference number, and a file reference number that is passed by reference as a returned value. It returns the error code as a function return value. The file reference number is used for subsequent calls to the opened file.

The routine "GetEOF" returns the number of bytes in the file. It expects two parameters: an integer that is a file reference number, and a long integer that returns (by reference) the number of valid bytes in the file. In our case, we place the long integer variable "charCount" in the second parameter. We assign this value to the ".TELength" field of our text record and use it to assign the size of the data accessed by the text handle field ".hText". We also pass it to the "FSRead" routine to determine how many bytes to read from the file.

The "FSRead" routine reads a specified number of bytes from a file and places them at a specified location in memory. It expects three parameters: an integer that holds the file reference number, a long integer that specifies the number of bytes to read, and a pointer that points to where the file bytes should be stored. The second parameter is passed by reference. If the routine is unable to read the entire amount specified in this parameter, then, upon return, this parameter contains the actual number of bytes read.

The last step in accessing the file is to close it with the "FSClose" routine. The "FSClose" routine expects one parameter, which is an integer containing the file's reference number.

If any of these calls fail, the "OpenFile" procedure aborts. If all are successful, then we get to the last part of the routine, where we call "TECalText" to calculate all positions of the "line starts" of the text. Recall that these are the positions within the block of text that correspond to the beginnings of lines as they are displayed on the screen. We then update the file name in "FName" and the volume reference number in "VRefNum". Previously, these were held in the respective fields of "reply". Finally, we update the file state Boolean variables "present", "titled", and "modified", setting the first two to true and the last to false.

Save File Command

The "SaveFile" procedure implements the "Save" command of the File menu. Its local variable, "charCount", is a long integer that temporarily stores the number of bytes in the file. It attempts to open the current file, write the entire text to it, then close it.

The "SaveFile" procedure also has a local "FLCall" routine to handle errors. This is identical to the previous "FLCall" routine, except that the target of the "Exit" is now our "SaveFile" procedure.

The procedure assigns the length of the file (as stored in the ".TE-Length" field of the edit record) to the local variable "charCount". It then makes a series of calls to the File Manager to open, write, and close the file. Again, we use our "FLCall" procedure to intercept possible errors generated by each call to the File Manager.

We have studied the File Manager routines to open and close files. Let's now look at the routine "FSWrite" to write data to files. The "FSWrite" routine expects three parameters: an integer that specifies the reference number of the file, a long integer passed by reference that specifies the number of bytes to write, and a pointer to the area of memory from which the bytes are taken.

The routine finishes by setting the Boolean variable "modified" equal to false.

Save As File Command

The "SaveAsFile" procedure implements the "Save As..." command of the File menu. It has three local variables: "charCount" is a long integer that stores the number of bytes in the file, "reply" is of type "SFReply" and holds the results from the Package Manager's standard file selection routine, and "theErr" is of type "OSErr" and temporarily stores the error code.

Our procedure has one subprocedure, "FLCall", to handle errors. It is similar to the "FLCall" subprocedures of the "OpenFile" and "SaveFile" procedures discussed earlier. Again, the difference is that the target of the "Exit" is the current procedure, "SaveAsFile".

The "SaveAsFile" procedure calls the routine "SFPutFile", which uses a dialog to get a specified file from the user. The "SFPutFile" is part of the Standard File Package.

This "SFPutFile" routine expects five parameters. The first parameter is a point that determines where the upper left corner is placed. The second parameter is a string that is displayed as a prompt in the dialog. The third parameter is a default file name that appears in the editable text

box when the dialog appears. The fourth parameter is a procedure pointer to a filter procedure for the dialog. The fifth parameter is a reply record of type "SFReply".

In our program, we pass the global point "where" as the first parameter, the string "Save text as:" as the second parameter, "fName" (the current name of the file) as the third parameter, NIL as the fourth parameter (no filter), and "reply" as the fifth parameter.

If the ".good" field of the "reply" record is true, indicating that a valid file is selected, we attempt to save the file. The file may not exist or it may differ from the current file, so the rules for saving the file are different. First, call "FSOpen" to find out whether the file exists or needs to be created. The error code indicates that the file does not yet exist, so we call "FCreate" to create it and "FSOpen" to open it. If the file already exists, we pass the error code to our local "FLCall" procedure. We then proceed to write the bytes to the file and close it as before. We use "FLCall" to catch any errors from each File Manager routine.

Once the file is properly saved, we load the new file name and volume reference number into the variables "fName" and "vRefNum", which hold the current values of these quantities. Notice that we use the values of these quantities obtained from "reply" to open our file. Thus, if the file saving is unsuccessful, the original file name and volume reference number remain.

We also load the new file name as the new window title and set the Boolean variable "titled" equal to true and the Boolean variable "modified" equal to false.

This file activity and this last updating activity are performed if the reply is good. We indent the entire section of code to indicate that it belongs in the THEN clause for "IF reply.good".

Close File Command

The "CloseFile" procedure implements the "Close" command of the File menu. It tries to save the file if it is modified, then disposes of the text record and hides the text window.

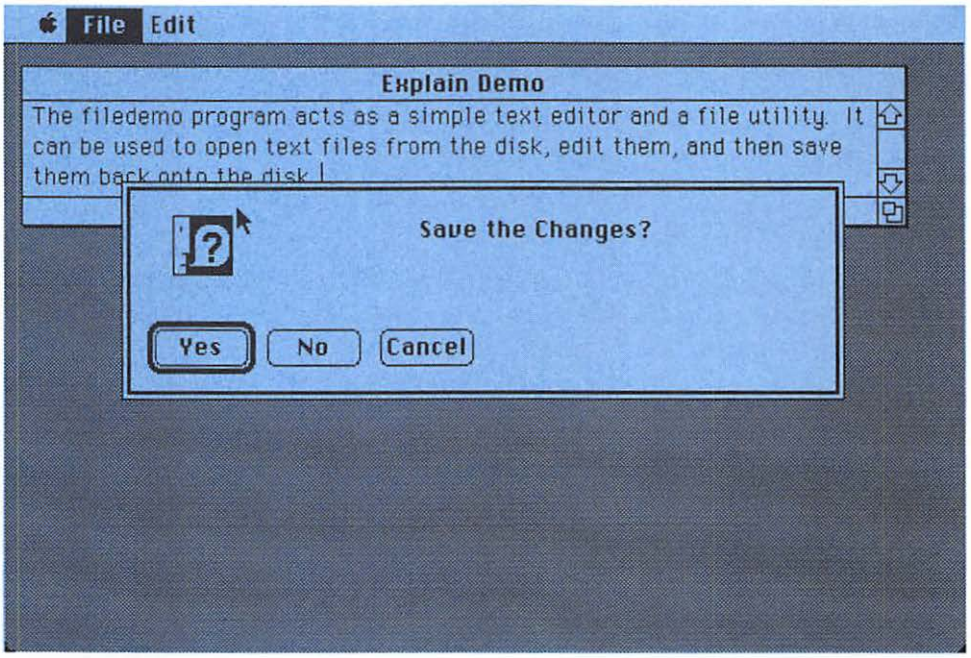
"CloseFile" resides at a higher level than previous file commands; that is, it calls them to save any files.

The procedure first checks "modified". If "modified" is true, then it calls a "caution" alert to see if the user wants to save the file or cancel

the "Close" command (see Figure 10-14). The caution alert is defined in the resource definition file as follows:

```
Type ALRT
  ,1004
  100 70 200 440
  1004
  7654

Type DITL
  ,1004
  4

  BtnItem Enabled
  70 10 90 60
Yes

  BtnItem Enabled
  70 130 90 180
Cancel
```

Figure 10-14. File Save Caution Alert



```
  BtnItem Enabled
  70 70 90 120
No

  StatText Disabled
  10 150 50 360
Save the Changes?
```

If the user wants to save the file, we check "titled" to see if we should use "SaveFile" or "SaveAsFile" to save it. If the user presses the cancel button, we "Exit" the "CloseFile" command.

After these preliminaries, we call "HideWindow" to make the window disappear and "TEDispose" to dispose of the text edit record, then set the Boolean variables "present" and "titled" to false.

Quit Command

The "QuitFile" procedure implements the "Quit" command of the File menu. It causes the program to terminate. However, it first tries to close the file, terminating only if the file successfully closes.

This procedure is actually a level above the "CloseFile" routine. It checks the Boolean variable "present" for a text window. If "present" indicates a text window, it calls "CloseFile" to attempt to close it. Then it sets "done" equal to "NOT present" so that the program terminates if "CloseFile" is successful or if there is no window to close.

Higher Level Initialization

The procedure "SetUpSys" performs all program initialization. It is called from the main program.

The "SetUpSys" procedure calls the standard initialization routines for each manager. It calls the "SetLimits", "SetUpWindows", "SetUpMenus", and "NewFile" procedures. We then initialize the cursor and set the global Boolean variable "done" equal to false.

Higher Level Updating

The procedure "UpdateSys" performs all updating in the main REPEAT loop.

It calls "SystemTask", which allows active desk accessories to update themselves. It then calls our "UpdateFState" routine to update the enable/disable state of the file menu items.

If "present" is true, it updates the text by calling "TEIdle" to allow the text caret to blink. The caret blinks at a fixed rate if this "idle" routine is called often enough. We also call our "UpdateScroll" routine if "present" is true. This updates the vertical scroll bar limit according to the current size of the text.

Implementing Menus

The next procedures implement our four menus. Except for the Apple menu, which involves desk accessories, they have a standard CASE structure: the routine that implements each menu item falls under each case. The procedure "DoAppleMenu" is nearly identical to the procedure of the same name in the example program for menus in Chapter 9.

For completeness, here are the resource definitions associated with the "About..." alert:

```
Type ALRT
  ,1001
  100 70 200 450
  1001
  4444

Type DITL
  ,1001
  3

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 10 30 370
FileDemo, a demonstration program for text and files

  StatText Disabled
  30 10 50 360
Christopher L. Morgan, 1985
```

Following these menu routines is a master menu selection routine.

File Menu

The procedure "DoFileMenu" implements the File menu. It has one parameter, an integer that specifies the menu item number. It consists of a CASE statement OF the item number parameter. The cases list the imple-

mentation procedures "NewFile", "OpenFile", "CloseFile", "SaveFile", "SaveAsFile", and "QuitFile". Notice that we skip item six because it is a disabled dashed line that separates "Quit" from the rest.

A real application would also have items in this menu to control the printing of files, but that is a more advanced topic not covered here.

Edit Menu

The procedure "DoEditMenu" implements the Edit menu. It has one parameter, the integer "theItem", which specifies the menu item number.

Our editing menu has only three commands: "Cut", "Copy", and "Paste". Many edit menus have more items. The standard menu has an "Undo" item, a unused item, then "Cut", "Copy", "Paste", and finally "Clear". This places our commands in the third, fourth, and fifth positions.

Now let's examine the procedure. It passes the editing command to a routine called "SystemEdit", which in turn passes editing commands to desk accessories. We add one to "theItem" to bring the "Cut", "Copy", and "Paste" command codes to two, three, and four (the standard positions starting the count from zero) before passing "theItem" to the "SystemEdit" routine.

If "SystemEdit" does not treat the edit command as an action on a desk accessory, it returns a value of false. In this case, we set the grafPort to the text window, set the Boolean variable "modified" equal to true if the item is "Cut" or "Paste", and execute a CASE statement to perform the appropriate "Cut", "Copy", or "Paste" command on the text window. Editing desk accessories and editing the text in the window are independent operations. The choice is determined by which window is currently selected.

Selecting Menus

The procedure "SelectMenu" is a general menu selection procedure called from the main event loop. It provides the proper structure for transmitting the selection information to the procedures that implement the various menus.

The "SelectMenu" procedure has one parameter, a long integer that contains both the menu and menu item selection information. The upper word is an integer that contains the resource identification number of the menus; the lower word is an integer that contains the item number of the item within that menu.

The procedure consists essentially of a CASE statement driven by the upper word of the selection information that selects the particular menu.

Each case in this CASE statement is a call to a procedure to handle the corresponding menu. These are "DoAppleMenu", "DoFileMenu", and "DoEditMenu", discussed previously. In each case, we pass the lower word (containing the item number) of the selection information to the individual menu procedure.

After the CASE statement, we call "HiliteMenu" to unhighlight the selected menu.

Dragging Windows

"WindowDrag" is a procedure called from the main event loop to handle the dragging of windows. It has one parameter, a point that specifies where the mouse cursor is when the mouse button is pressed.

The "WindowDrag" procedure calls the Window Manager's "DragWindow" procedure, passing the appropriate parameters to it. It makes the main event loop appear more compact and readable.

Scroll Action

The procedure "ScrAction" is an action procedure for scrolling. Action procedures were introduced in Chapter 6.

The procedure has two parameters: a control handle to specify a particular control and a part code to specify a particular part of that control. It has four local variables. It has two integers: "pageSize" specifies the number of text lines in the currently sized text window, and "delta" specifies how many lines to scroll. The procedure also has two points, "S" and "dS", which hold the amounts to be scrolled as they are computed.

The procedure first computes "pageSize" by dividing the height of the view rectangle by the height of a line of text. It uses information in the text record "theTE". Next, it executes a CASE statement to find which buttonlike part of the scroll bar is selected. For the up and down buttons, we select a value of minus and plus one; for the up and down page selectors, we select minus or plus the "pageSize". The appropriate value is placed in "delta". If none of these cases is selected, we immediately exit.

Once the raw amount to scroll is selected, we use the scroll control to determine the actual amount to scroll. This clips values that are beyond the scrolling limits. We temporarily store the current scroll value in "S", use "delta" to update the scroll value, then place the difference between the original value and the new value in dS. We pass dS times the line height to the scrolling routine "TEScroll". Because we store the number

of text lines (not the number of pixels) in the scroll control, we always scroll by a whole number of lines.

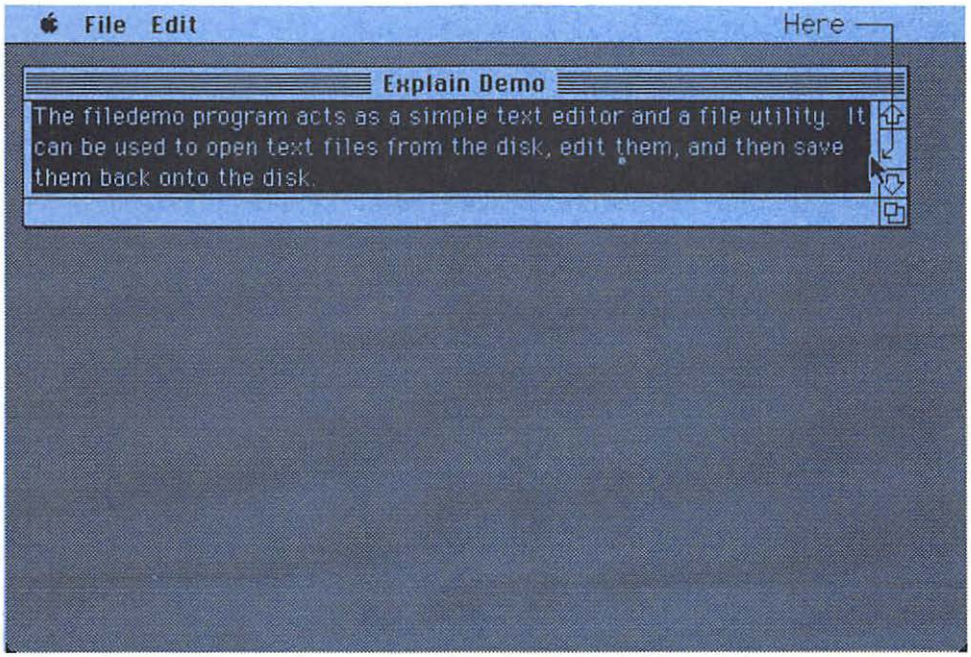
The Text Edit scroll routine, "TEScroll", expects three parameters: the horizontal amount to scroll, the vertical amount to scroll, and a handle to the text edit record. In our program, we pass zero for the horizontal scrolling amount and "theTE" for the text edit handle.

Controlling the Window

The procedure "WindowControl" is called from the main event loop to handle mouse events that occur within a window's content area, including its scroll bars.

Two main cases arise: the selected window is in front (selected or highlighted window) or it is not. If the selected window is in front, we must transform the mouse point to local coordinates and look at two cases: the mouse point is in the text viewing area or is not (see Figure 10-15). If the mouse point falls within the text viewing area, we call Text Edit's "TEClick" routine to allow text selection.

Figure 10-15. Where's the Mouse?



The "TEClick" procedure has three parameters: a point that specifies the mouse position, a Boolean that specifies if we are in the extended mode (the shift key is down during selection), and a handle to the text record. In this program, we use the "BitTst" routine to test the shift key bit in the ".modifier" field of the event record.

If the mouse point is not in the text viewing area, we call "FindControl" to find a control that it might be in. In particular, we want to see if it is in the vertical scroll control bar. The basic structure of this section is explained in Chapter 6.

If the mouse point is not in the front window, we call "SelectWindow" and a couple of other routines to make it the front window, as in Chapter 7.

Growing Windows

The procedure "WindowGrow" is called from the main event loop to resize the window when the grow icon is selected. It is much the same as the "WindowGrow" procedure introduced in Chapter 6. However, it resizes one window (our text window) with only one scroll control (the vertical scroll bar). To simplify programming, it causes the entire port rectangle to update each time the window is resized. It also resizes the viewing area according to the new window size. The viewing area is slightly indented within the center of the window (see Figure 10-16).

Handling Key Events

The procedure "KeyEvent" is called from the main event loop to handle keyboard events. It has one parameter, of type CHAR, that specifies which key was hit.

The "KeyEvent" procedure checks to see if the command key is down. We use the "BitTst" routine to test bit seven (from the left) of the ".modifiers" field of the event record.

If the command key is down, we call the Menu Manager's "MenuKey" to map the key into menu selection information, then our "SelectMenu" routine to handle the selection. In our program, this takes care of the command key alternatives: "Q" for "Quit", "X" for "Cut", "C" for "Copy", and "V" for "Paste".

If the command key is not selected, we see if the text window is the front window and the text record is open. If these are both true, then we call Text Edit's "TEKey" to insert the key character into the text and display the newly modified text on the screen. We also set the Boolean variable "modified" equal to true to indicate that the text has been modified.

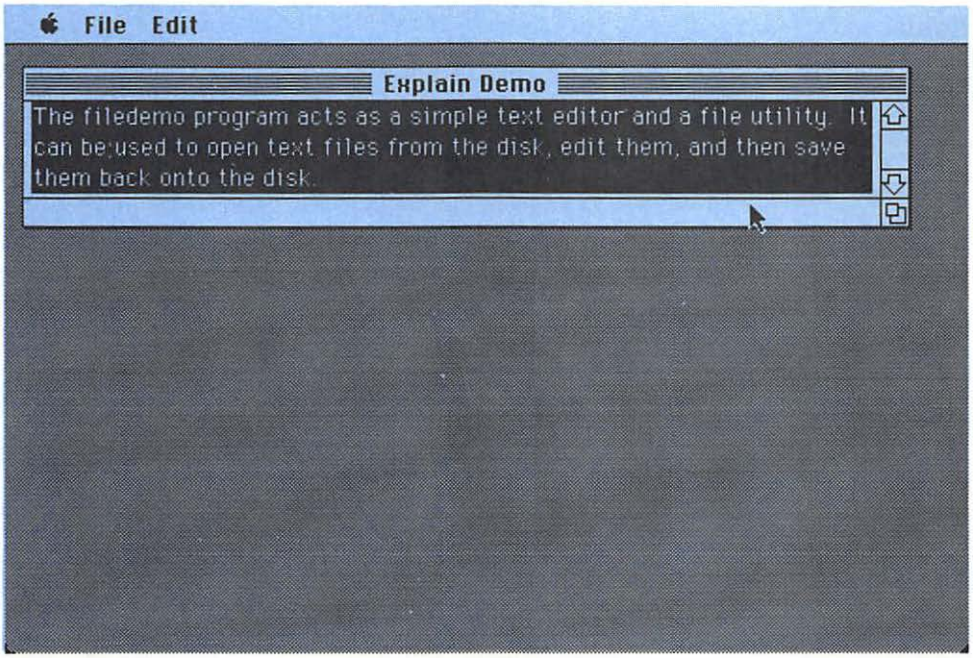
At this point in the program, you can add code to provide automatic scrolling as the text caret slips below the viewing area of the text window. This code must compute the lines in the window using the size of the window and the size and vertical separation of the text. Then it must compare the line starts with the selection range to determine where the caret is in relation to the page. Finally, it must call for the proper amount of scrolling to keep the caret within the window.

Updating Windows

The procedure "WindowUpdate" is called from the main event loop to update a window in response to an update event. It works much the same as in previous chapters, except that we call Text Edit's "TEUpdate" to update the text between the "BeginUpdate" and "EndUpdate" routines. However, we call it only if the window that we are updating is the text window and the text record is open.

The "TEUpdate" routine expects two parameters: a rectangle that specifies where we need to update, and a handle to the text record. In our

Figure 10-16. The Text Viewing Area (in black)



program, we pass the region boundary box of the visible region of the window as the update rectangle, and we pass "theTE" as the text handle.

Activating Windows

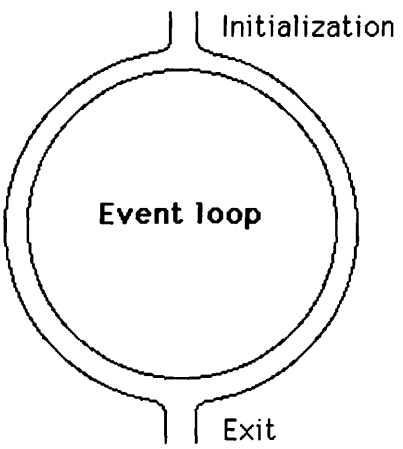
The procedure "WindowActivate" is called from the main event loop in response to an activate event. It first calls our "WindowUpdate" routine, then calls "TEActivate" to activate or "TEDeActivate" to deactivate the text, depending on the lowest order bit of the ".modifiers" field of the event record. This bit specifies whether the activate event indicates activation or a deactivation.

The Main Program

The main program is a generic event loop with an initialization section (see Figure 10-17). Each major part of this structure is encapsulated in a procedure assigned to perform a specific function in relation to the event loop. Many useful applications programs have this basic structure and could use this same main program.

The first step of the main program is a call to "SetUpSys" which performs the entire initialization of the program. Our procedure "SetUpSys" intializes a number of different managers and sets up our menus and main window as well as some global limits, such as the drag and size limits.

Figure 10-17. Event Loop with Initialization



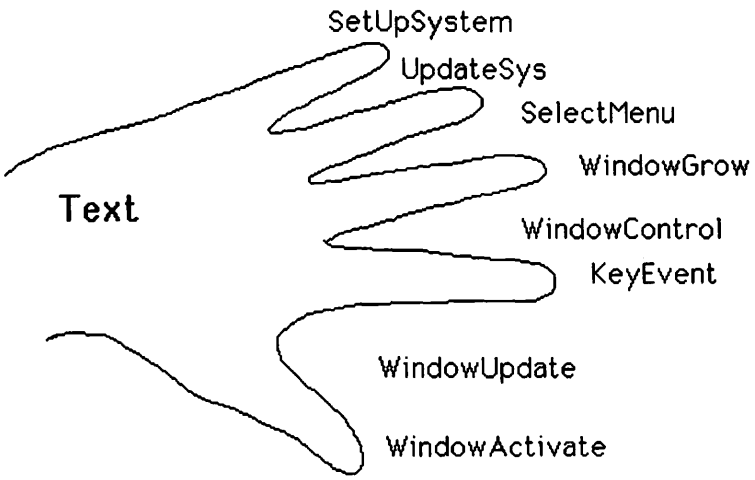
Next, the main REPEAT..UNTIL loop begins with a call to "UpdateSys". This is a different initialization procedure, done each time through the loop. Thus, we call it an update routine. In our program, "UpdateSys" allows such actions as a slice of background task for the desk accessories, the blinking of the text caret, updating scrolling limits, and updating the enable/disable state of the File menu.

After the update procedure, we call "GetNextEvent" in an IF..THEN statement. Within the THEN clause, we have a CASE statement driven by the ".what" field of the event record. The cases under this CASE statement form a list of the kinds of events tracked by our program. In this program, we track mouse down, key down, auto key, update, and activate.

Within mouse down, we have a CASE statement that sorts where the mouse was when its button was pressed. The cases are in the menu bar, in a system window such as a desk accessory, in the drag region of a window, in the contents region of a window, and in the grow region of a window. In each case, we call a routine to perform a particular action on whichever window is there.

Files and text are not manifested directly in the main program; yet they affect many of the procedures that are called by the main program. This applies particularly to text. That is, the text routines of Text Edit act like a number of fingers that must be inserted in just the right places to make a piece of machinery (our program) work properly (see Figure 10-18).

Figure 10-18. Text Edit's "Figures"



In the case of files, the initialization procedure "SetUpSys" must initialize the File and Information menus and some file variables, the update procedure "UpdateSys" must update the File menu, and the menu select procedure "SelectMenu" must list the File and Information menus.

In the case of text, much more is required. Almost every procedure called from the main program must perform a vital text task. For example, text is initialized in "SetUpSys", sized in "UpdateSys", edited and loaded and saved into files within "SelectMenu", scrolled within "WindowControl", activated in window resizing in "WindowGrow", edited within "KeyEvent", and redrawn within "WindowUpdate" and "WindowActivate". This makes programming text especially difficult, because of the interrelationship of statements that are scattered throughout the program.

Summary

In this chapter, we have studied files and text. We have seen how they can be programmed into useful applications, such as a text editor or a file utility. We have seen how they relate to each other and to the basic structure of an event-driven applications program.

The following ROM routines are covered in this chapter:

ROM Routines

MN-EnableItem

MN-DisableItem

CM-SetCtlMax

PK-GetIndStr

PK-NumToString

DL-ParamText

TE-TENew

WM-SetWTitle

PK-SFGetFile

FL-FSOpen

FL-GetEOF

MM-SetHandleSize

FL-FSRead

FL-FSClose

TE-TECalText

FL-FSWrite
PK-SFPutFile
FL-Create
TE-TEDispose
FL-GetVInfo
FL-Eject
FL-GetFInfo
FL-SetFInfo
TE-TEIdle
DS-SystemEdit
TE-TECut
TE-TECopy
TE-TEPaste
TE-TEScroll
TE-TEClick
TU-BitTst
TE-TEKey
TE-TEUpdate
TE-TEActivate
TE-TEDeactivate

A

ROM Routines Sorted by Name

Appendix A contains a table of the ROM routines sorted by name in alphabetical order. The first column contains the instruction number (see Chapter 2), the second column contains the address in ROM or RAM where that routine is currently located, the third column contains the name of the routine, and the fourth column contains the manager to which it belongs and the page number in the edition of *Inside the Apple Macintosh* (Cupertino: Apple Computer, Inc., 1985) used during the development process.

If you type this list into your computer, update the last column to match your edition of *Inside the Apple Macintosh*. You then can sort it by columns and print out these sorted listings for your own reference.

The first few entries are instruction numbers that are not used and therefore have no names.

A list of Apple's abbreviations for the various managers is included.

Table A-1. List of Abbreviations

Abbreviation	*Manager*
CM	Control Manager
DL	Dialog Manager
DS	Desk Manager
DV	Device Manager
EM	Event Manager

Abbreviation	Manager
FL	File Manager
MM	Memory Manager
MN	Menu Manager
OU	Operating System Utilities
PK	Package Manager
QD	QuickDraw
RM	Resource Manager
SL	Segment Loader
SM	Scrap Manager
TE	Text Edit
TU	Toolbox Utilities
VR	Vertical Retrace Manager
WM	Window Manager

Table A-2. List of Routines

Number	Address	Name	Module
060	40 0594		
08F	40 0594		
095	40 0594		
09F	40 0594		
0B5	40 0594		
0D7	40 0594		
0F7	40 0594		
0FD	40 0594		
152	40 0594		
153	40 0594		
16D	40 0594		
178	40 0594		
184	40 0594		
1C3	40 0594		
1C4	40 0594		
1C5	40 0594		
1F8	40 0594		
1FF	40 0594		
04E	40 174A	AddDrive	OS3.2-52
07E	40 5BB4	AddPt	QD-65

Number	Address	Name	Module
1AC	40 E21C	AddReference	RM-26
14D	40 D072	AddResMenu	MN-16
1AB	40 E14A	AddResource	RM-25
185	40 E4B8	Alert	DL-23
010	00 2180	Allocate	FL-21,FL-44
0C4	40 89CC	AngleFromSlope	
133	40 CEE8	AppendMenu	MN-17
063	40 5486	BackColor	QD-46
07C	40 5BA0	BackPat	QD-39
122	40 C2EE	BeginUpdate	WM-32
058	40 53B4	BitAnd	TU-08
05F	40 541A	BitClr	TU-07
05A	40 53CA	BitNot	TU-08
05B	40 53D2	BitOr	TU-08
05E	40 5408	BitSet	TU-07
05C	40 53DC	BitShift	TU-08
05D	40 53F2	BitTst	TU-07
059	40 53BE	BitXor	TU-08
02E	00 26B0	BlockMove	MM-47
120	40 C1FA	BringToFront	WM-25
174	40 B6AC	Button	EM-19
148	00 27F0	CalcMenuSize	MN-26
109	40 B8F6	CalcVis	WM-36
10A	40 B94E	CalcVisBehind	WM-37
188	40 E4B4	CautionAlert	DL-24
1F3	40 4C0C	Chain	SL-06
1AA	00 2226	ChangedResData	RM-24
08D	40 5E2C	CharWidth	QD-44
145	40 CDA6	CheckItem	MN-23
111	40 BBA4	CheckUpdate	WM-35
134	40 C7A0	ClearMenuBar	MN-19
10B	40 B992	ClipAbove	WM-31
07B	40 5B8A	ClipRect	QD-38
001	40 138A	Close	FL-22,FL-45,DV-07,DV-14
1B7	40 F1F4	CloseDeskAcc	DS-07
182	40 E7B0	CloseDialog	DL-21
0F4	40 A69C	ClosePicture	QD-62
0CC	40 8B94	ClosePoly	QD-63
07D	40 59E2	ClosePort	QD-36
19A	00 2376	CloseResFile	RM-16
0DB	40 8EEA	CloseRgn	QD-56
12D	00 21A2	CloseWindow	WM-22

Number	Address	Name	Module
03C	40 4E0E	CmpString	OU-12
064	40 548C	ColorBit	QD-46
04C	40 2BA0	CompactMem	MM-39
004	00 291C	Control	DV-08,DV-17
0EC	40 9BA4	CopyBits	QD-60
0DC	40 8F42	CopyRgn	QD-55
189	40 E84A	CouldAlert	DL-25
179	00 23DA	CouldDialog	DL-23
150	40 D142	CountMItems	MN-26
19C	40 DAE8	CountResources	RM-19
19E	40 DB40	CountTypes	RM-18
008	40 3FBC	Create	FL-18,FL-37
1B1	40 D876	CreateResFile	RM-16
194	40 E096	CurResFile	RM-18
1C7	40 FF96	Date2Secs	OU-15
03B	40 4DFC	Delay	OU-22
009	40 408E	Delete	FL-24,FL-51
136	40 C84E	DeleteMenu	MN-18
14F	40 BF80	DeltaPoint	
16E	40 0B44	Dequeue	OU-19
192	40 E018	DetachResource	RM-22
180	40 E6DE	DialogSelect	DL-21
0E6	40 913C	DiffRgn	QD-57
13A	40 C9AC	DisableItem	MN-22
155	40 D2A6	DisposControl	CM-16
183	40 E81E	DisposDialog	DL-23
023	40 2CC8	DisposHandle	MM-31
01F	40 2C6A	DisposPtr	MM-35
0D9	40 8EA6	DisposRgn	QD-54
114	00 21AA	DisposWindow	WM-23
132	40 CCD2	DisposeMenu	MN-16
167	40 D4B2	DragControl	CM-21
105	40 C424	DragGrayRgn	WM-33
126	40 C430	DragTheRgn	TU-07 (WM-30 called DragGreyRgn)
125	40 C36A	DragWindow	WM-28
083	40 5DEC	DrawChar	QD-44
169	40 D648	DrawControls	CM-18
181	40 E79A	DrawDialog	DL-23
104	40 C746	DrawGrowIcon	WM-26
137	00 258C	DrawMenuBar	MN-18
10F	40 BAFC	DrawNew	WM-36
0F6	40 A6E8	DrawPicture	QD-62

Number	Address	Name	Module
084	40 5DFE	DrawString	QD-44
085	40 5E12	DrawText	QD-44
03D	40 14A6	DrvrInstall	
03E	40 14F4	DrvrRemove	
017	00 20FA	Eject	FL-17,FL-36
02B	40 2D0A	EmptyHandle	MM-41
0AE	40 7160	EmptyRect	QD-48
0E2	40 90E0	EmptyRgn	QD-58
139	40 C990	EnableItem	MN-23
123	40 C328	EndUpdate	WM-32
16F	40 0B20	Enqueue	OU-19
081	40 5BFA	EqualPt	QD-65
0A6	40 7146	EqualRect	QD-48
0E3	40 90F0	EqualRgn	QD-58
0C0	40 8038	EraseArc	QD-53
0B9	40 7E2C	EraseOval	QD-50
0C8	40 8B3A	ErasePoly	QD-65
0A3	00 1E8A	EraseRect	QD-49
0D4	40 8DAE	EraseRgn	QD-59
0B2	40 7D52	EraseRoundRect	QD-51
18C	40 E8F6	ErrorSound	DL-18
171	40 B75E	EventAvail	EM-18
1F4	00 205C	ExitToShell	SL-07
101	00 25B6	FMSwapFont	FM-11
0C2	40 8044	FillArc	QD-54
0BB	40 7E38	FillOval	QD-50
0CA	40 8B46	FillPoly	QD-65
0A5	40 6FF0	FillRect	QD-49
0D6	40 8DBA	FillRgn	QD-59
0B4	40 7D5E	FillRoundRect	QD-52
16C	40 D6B6	FindControl	CM-19
12C	40 C6C4	FindWindow	WM-26
068	40 57DC	FixMul	TU-04
069	00 25C0	FixRatio	TU-04
06C	40 5854	FixRound	TU-04
14C	00 2670	FlashMenuBar	MN-26
032	40 389A	FlushEvents	EM-19,OSEM-04
045	40 4796	FlushFile	FL-45
013	40 3C94	FlushVol	FL-17,FL-34
062	40 5480	ForeColor	QD-45
0BE	40 802C	FrameArc	QD-52
0B7	40 7E20	FrameOval	QD-50

Number	Address	Name	Module
0C6	40 8B2E	FramePoly	QD-64
0A1	40 6FD8	FrameRect	QD-49
0D2	40 8DA2	FrameRgn	QD-58
0B0	40 7D46	FrameRoundRect	QD-51
18A	40 E88E	FreeAlert	DL-25
17A	00 2406	FreeDialog	DL-23
01C	40 2BE0	FreeMem	MM-38
124	00 25F8	FrontWindow	WM-26
1F5	40 4D30	GetAppParms	SL-06,ST-09
15A	40 D350	GetCRefCon	CM-25
15E	40 D3CA	GetCTitle	CM-19
07A	40 5B76	GetClip	QD-38
16A	40 D36E	GetCtlAction	CM-24
160	40 D424	GetCtlValue	CM-23
1B9	40 F228	GetCursor	TU-09
18D	40 E8FE	GetDItem	DL-26
011	40 483E	GetEOF	FL-20,FL-43
0FF	40 B5B2	GetFName	FM-10
100	40 B5F0	GetFNum	FM-10
018	40 448C	GetFPos	FL-20,FL-42
00C	40 4390	GetFileInfo	FL-22,FL-46
08B	40 600E	GetFontInfo	QD-45
025	00 2710	GetHandleSize	MM-31
190	40 E972	GetIText	DL-27
1BB	40 F238	GetIcon	TU-07
19D	40 DB24	GetIndResource	RM-19
19F	40 DB88	GetIndType	RM-18
146	40 CE46	GetItem	MN-22
13F	40 CD8E	GetItmIcon	MN-24
143	40 CD9E	GetItmMark	MN-25
141	40 CD96	GetItmStyle	MN-24
176	40 B690	GetKeys	EM-20
149	40 D004	GetMHandle	MN-26
162	40 D436	GetMaxCtl	CM-23
13B	40 CCAE	GetMenuBar	MN-19
161	40 D432	GetMinCtl	CM-23
172	40 B702	GetMouse	EM-19
1A1	40 DC0C	GetNamedResource	RM-20
1BE	40 F29E	GetNewControl	CM-18
17C	40 E5D4	GetNewDialog	DL-21
1C0	40 F32E	GetNewMBar	MM-19
1BD	40 F248	GetNewWindow	WM-22

Number	Address	Name	Module
170	00 241E	GetNextEvent	EM-17
031	40 387C	GetOSEvent	OSEM-04
1B8	40 F20E	GetPattern	TU-09
09A	40 68A8	GetPen	QD-40
098	40 6882	GetPenState	QD-41
1BC	40 F240	GetPicture	TU-10
065	40 564A	GetPixel	QD-68
074	40 5AA4	GetPort	QD-36
021	40 2C74	GetPtrSize	MM-36
1BF	40 F2EE	GetRMenu	MM-16
1A6	40 E09E	GetResAttrs	RM-22
1F6	40 D8EA	GetResFileAttr	RM-29
1A8	40 E0DC	GetResInfo	RM-22
1A0	40 DBE2	GetResource	RM-20
1FD	40 FDDC	GetScrap	SM-12
1BA	40 F230	GetString	TU-04
046	40 108E	GetTrapAddress	OU-21
014	40 3E40	GetVol	FL-16,FL-33
007	40 3E74	GetVolInfo	FL-16,FL-32
110	40 BB9A	GetWMgrPort	WM-21
117	40 BEE0	GetWRefCon	WM-33
119	40 BF0A	GetWTitle	WM-23
12F	40 BF04	GetWindowPic	WM-33
01A	40 2B92	GetZone	MM-29
071	40 5A74	GlobalToLocal	QD-66
072	40 5A94	GrafDevice	QD-36
12B	40 C5BE	GrowWindow	WM-29
029	40 2D44	HLock	MM-42
04A	40 2D68	HNoPurge	MM-43
049	40 2D5C	HPurge	MM-43
02A	40 2D50	HUnlock	MM-42
1E4	40 EF16	HandAndHand	OU-11
1E1	00 2694	HandToHand	OU-09
026	40 2CF6	HandleZone	MM-33
06A	40 5840	HiWord	TU-06
158	40 D2F8	HideControl	CM-17
052	40 5366	HideCursor	QD-39
096	40 686E	HidePen	QD-40
116	40 BEB6	HideWindow	WM-23
15D	40 D39E	HiliteControl	CM-18
138	40 C966	HiliteMenu	MN-21
11C	40 C0D2	HiliteWindow	WM-25

Number	Address	Name	Module
1A4	40 E080	HomeResFile	RM-18
1F9	40 FCE2	InfoScrap	SM-10
1E6	00 262A	InitAllPacks	PK-05
02C	00 201A	InitApplZone	MM-25
050	40 533A	InitCursor	QD-39
17B	40 E440	InitDialogs	DL-18
0FE	00 2598	InitFonts	FM-09
06E	00 2566	InitGraf	QD-34
130	40 C764	InitMenus	MN-15
1E5	40 FE82	InitPack	PK-05
06D	40 5962	InitPort	QD-35
016	40 39A0	InitQueue	FL-31
195	40 D744	InitResources	RM-15
03F	40 4F5A	InitUtil	OU-17
112	00 2578	InitWindows	WM-20
019	00 285C	InitZone	MM-27
135	40 C7AC	InsertMenu	MN-18
151	40 D07A	InsertResMenu	MN-18
0A9	00 2554	InsetRect	QD-47
0E1	40 903A	InsetRgn	QD-57
128	40 C598	InvalRect	WM-31
127	40 C558	InvalRgn	WM-32
0D5	40 8DB4	InverRgn	QD-59
0C1	40 803E	InvertArc	QD-54
0BA	40 7E32	InvertOval	QD-50
0C9	40 8B40	InvertPoly	QD-65
0A4	40 6FEA	InvertRect	QD-49
0B3	40 7D58	InvertRoundRect	QD-52
17F	40 E6A4	IsDialogEvent	DL-20
156	40 D2C8	KillControls	CM-17
006	40 1468	KillIO	DV-10,DV-18
0F5	40 A6E4	KillPicture	QD-62
0CD	40 8C02	KillPoly	QD-63
1F2	40 4C12	Launch	SL-07
092	40 67BE	Line	QD-42
091	40 67A8	LineTo	QD-42
06B	40 584A	LoWord	TU-06
1A2	40 DFCA	LoadResource	RM-20
1F0	40 4B44	LoadSeg	SL-08
070	40 5A54	LocalToGlobal	QD-66
1FB	40 FD66	LodeScrap	SM-11
067	40 5776	LongMul	TU-07

Number	Address	Name	Module
0FC	40 8C22	MapPoly	QD-70
0F9	40 B18E	MapPt	QD-69
0FA	40 B1F0	MapRect	QD-69
0FB	40 9396	MapRgn	QD-69
01D	40 2C08	MaxMem	MM-38
13E	40 CDBA	MenuKey	MN-21
13D	40 C9E2	MenuSelect	MN-20
191	40 E71E	ModalDialog	DL-21
036	00 273C	MoreMasters	
00F	00 20BA	MountVol	FL-31
094	40 67E2	Move	QD-42
159	40 D312	MoveControl	CM-21
077	40 5AE8	MovePort	QD-37
093	40 67D6	MoveTo	QD-42
11B	40 BF9A	MoveWindow	WM-28
1E0	00 2806	Munger	TU-05
154	40 D19A	NewControl	CM-15
17D	40 E60C	NewDialog	DL-20
022	00 292C	NewHandle	MM-30
131	40 CE9E	NewMenu	MN-15
01E	40 2C4E	NewPtr	MM-35
0D8	40 8E82	NewRgn	QD-54
106	40 B868	NewString	TU-04
113	40 BCE8	NewWindow	WM-21
187	40 E4B0	NoteAlert	DL-24
030	40 37F4	OSEventAvail	OSEM-03
056	40 53B0	ObscureCursor	QD-40
035	00 20F4	Offline	FL-35
0CE	40 8C06	OffsetPoly	QD-63
0A8	40 717A	OffsetRect	QD-46
0E0	40 9008	OffsetRgn	QD-56
000	40 1262	Open	FL-18,FL-38,DV-07,DV-14
1B6	40 F1A8	OpenDeskAcc	DS-07
0F3	40 A5EE	OpenPicture	QD-61
0CB	40 8B5E	OpenPoly	QD-62
06F	40 594A	OpenPort	QD-35
00A	40 3F04	OpenRF	FL-39
197	00 21CA	OpenResFile	RM-16
0DA	40 8EBA	OpenRgn	QD-55
1E7	40 FF0E	Pack0	PK-04
1E8	40 FF10	Pack1	PK-04
1E9	40 FF12	Pack2	PK-04

Number	Address	Name	Module
1EA	40 FF14	Pack3	PK-04
1EB	40 FF16	Pack4	PK-04
1EC	40 FF18	Pack5	PK-04
1ED	40 FF1A	Pack6	PK-04
1EE	40 FF1C	Pack7	PK-04
0CF	40 9D42	PackBits	
0BF	40 8032	PaintArc	QD-53
10D	40 BA62	PaintBehind	WM-36
10C	40 B9B6	PaintOne	WM-36
0B8	40 7E26	PaintOval	QD-50
0C7	40 8B34	PaintPoly	QD-64
0A2	40 6FDE	PaintRect	QD-49
0D3	40 8DA8	PaintRgn	QD-59
0B1	40 7D4C	PaintRoundRect	QD-51
18B	40 E8D2	ParamText	DL-25
09C	40 68C6	PenMode	QD-41
09E	40 68E0	PenNormal	QD-42
09D	40 68CC	PenPat	QD-42
09B	40 68B8	PenSize	QD-41
0F2	40 A5D8	PicComment	QD-62
14E	40 1006	PinRect	WM-33,TU-07
14B	40 D15E	PlotIcon	TU-07
076	40 5AC8	PortSize	QD-37
02F	40 377C	PostEvent	EM-18,OSEM-03
0AC	40 7280	Pt2Rect	QD-47
0AD	40 72B6	PtInRect	QD-47
0E8	40 923E	PtInRgn	QD-58
0C3	40 89FE	PtToAngle	QD-48
1EF	40 DF1A	PtrAndHand	OU-11
1E3	40 EEFC	PtrToHand	OU-10
1E2	40 EEF4	PtrToXHand	OU-10
048	40 2C88	PtrZone	MM-37
04D	40 2BC4	PurgeMem	MM-40
1CA	40 0790	PutIcon	
1FE	40 FE3A	PutScrap	SM-13
04F	40 4D4C	RDrvrInstall	
061	40 542C	Random	QD-67
027	40 2D1A	ReAllocHandle	MM-34
002	00 1CB2	Read	FL-19,FL-40,DV-08,DV-15
039	40 4DEA	ReadDateTime	OU-14
037	40 4DA8	ReadParm	
102	40 B65A	RealFont	FM-10

Number	Address	Name	Module
028	40 2CFE	RecoverHandle	MM-33
0E9	40 929E	RectInRgn	QD-58
0DF	40 8FFA	RectRgn	QD-55
1A3	40 E008	ReleaseResource	RM-21
00B	40 4122	Rename	FL-23,FL-50
1AF	40 E334	ResError	RM-17
040	40 2BEA	ResrvMem	MM-39
1AE	40 E30C	RmveReference	RM-26
1AD	40 E288	RmveResource	RM-26
196	40 D81A	RsrcZoneInit	RM-15
042	40 4274	RstFilLock	FL-23,FL-48
10E	40 BACE	SaveOld	WM-36
0F8	40 B136	ScalePt	QD-68
0EF	40 9C40	ScrollRect	QD-59
1C6	40 FF1E	Secs2Date	OU-16
0AA	40 719E	SectRect	QD-47
0E4	40 9130	SectRgn	QD-57
17E	40 E9EC	SelIText	DL-27
11F	40 C1CA	SelectWindow	WM-23
121	40 C298	SendBehind	WM-25
057	40 2AE6	SetAppBase	MM-26
02D	00 2728	SetApplLimit	MM-28
15B	40 D35E	SetCRefCon	CM-24
15F	40 D3E2	SetCTitle	CM-18
079	40 5B66	SetClip	QD-38
16B	40 D372	SetCtlAction	CM-24
163	40 D43A	SetCtlValue	CM-22
051	40 5348	SetCursor	QD-39
18E	40 E93C	SetDItem	DL-26
03A	40 4DF4	SetDateTime	OU-15
012	40 48C0	SetEOF	FL-21,FL-43
0DD	40 8F90	SetEmptyRgn	QD-55
044	40 4490	SetFPos	FL-20,FL-42
041	40 426A	SetFilLOck	FL-23,FL-48
043	40 423E	SetFilType	FL-49
00D	40 429E	SetFileInfo	FL-22,FL-47
103	00 27DA	SetFontLock	FM-10
04B	00 2876	SetGrowZone	MM-44
024	40 2CE4	SetHandleSize	MM-32
18F	40 E992	SetIText	DL-27
147	40 D024	SetItem	MN-22
140	40 CD92	SetItmIcon	MN-23

Number	Address	Name	Module
144	40 CDA2	SetItmMark	MN-25
142	40 CD9A	SetItmStyle	MN-24
14A	40 D068	SetMFlash	MN-25
165	40 D480	SetMaxCtl	CM-23
13C	40 CCC4	SetMenuBar	MN-19
164	40 D47C	SetMinCtl	CM-23
078	40 5B18	SetOrigin	QD-38
075	40 5AB0	SetPBits	QD-37
099	40 6886	SetPenState	QD-41
073	40 5A9A	SetPort	QD-36
080	40 5BF0	SetPt	QD-65
020	40 2C7E	SetPtrSize	MM-37
0A7	40 7138	SetRect	QD-46
0DE	40 8F9C	SetRectRgn	QD-55
1A7	00 2254	SetResAttrs	RM-24
1F7	40 D8F4	SetResFileAttr	RM-29
1A9	00 2274	SetResInfo	RM-23
19B	40 DAE0	SetResLoad	RM-19
193	40 DAB8	SetResPurge	RM-28
0EA	40 59FE	SetStdProcs	QD-71
107	40 B880	SetString	TU-04
047	40 109C	SetTrapAddress	OU-21
015	40 3E2E	SetVol	FL-16,FL-33
118	40 BEEE	SetWRefCon	WM-33
11A	40 BF22	SetWTitle	WM-23
12E	40 BEFE	SetWindowPic	WM-33
01B	40 2B98	SetZone	MM-29
055	40 536E	ShieldCursor	TU-10
157	40 D2DA	ShowControl	CM-17
053	40 536A	ShowCursor	QD-39
108	40 BB5C	ShowHide	WM-24
097	40 6878	ShowPen	QD-40
115	40 BE90	ShowWindow	WN-21
15C	40 D376	SizeControl	CM-23
1A5	00 21FC	SizeResource	RM-01
11D	40 C112	SizeWindow	WM-30
0BC	40 8830	SlopeFromAngle	
08E	40 5DDE	SpaceExtra	QD-44
005	40 1444	Status	DV-09,DV-17
0BD	40 7F86	StdArc	QD-72
0EB	40 9A1E	StdBits	QD-72
0F1	40 A4DC	StdComment	QD-73

Number	Address	Name	Module
0EE	40 A544	StdGetPic	QD-73
090	40 6710	StdLine	QD-71
0B6	40 7D76	StdOval	QD-72
0C5	40 8A94	StdPoly	QD-72
0F0	40 A568	StdPutPic	QD-73
0AF	40 7C8A	StdRRect	QD-72
0A0	40 6F1C	StdRect	QD-72
0D1	40 8D18	StdRgn	QD-72
082	40 5C06	StdText	QD-71
0ED	40 5EBE	StdTxMeas	QD-73
173	40 B718	StillDown	EM-19
186	40 E4AC	StopAlert	DL-24
08C	40 5E46	StringWidth	QD-45
066	40 5686	StuffHex	QD-68
07F	40 5BD2	SubPt	QD-65
1C8	00 2644	SysBeep	OU-22
1C2	40 F17A	SysEdit	DS-08
1C9	40 0944	SysError	
1B3	00 27C6	SystemClick	DS-07
1B2	00 25DC	SystemEvent	DS-09
1B5	40 F132	SystemMenu	DS-10
1B4	40 F0BA	SystemTask	DS-08
1D8	40 FB1E	TEActivate	TE-18
1D0	40 F43C	TECalText	TE-19
1D4	40 F53E	TEClick	TE-17
1D5	40 F8BE	TECopy	TE-15
1D6	40 F8E2	TECut	TE-15
1D9	40 FB40	TEDeactivate	TE-18
1D7	40 F8F0	TEDelete	TE-16
1CD	40 F3C2	TEDispose	TE-14
1CB	40 F390	TEGetText	TE-14
1DA	40 FB56	TEIdle	TE-18
1CC	40 F39C	TEInit	TE-13
1DE	40 FBEA	TEInsert	TE-16
1DC	40 FC18	TEKey	TE-14
1D2	40 F4A2	TENew	TE-13
1DB	40 FB80	TEPaste	TE-15
1DD	40 FC56	TEScroll	TE-19
1DF	40 FC94	TESetJust	TE-17
1D1	40 F480	TESetSelect	TE-17
1CF	40 F416	TESetText	TE-14
1D3	40 F50A	TEUpdate	TE-18

Number	Address	Name	Module
166	40 D484	TestControl	CM-18
1CE	40 F3DA	TextBox	TE-19
088	40 5DC2	TextFace	QD-43
087	40 5DBC	TextFont	QD-43
089	40 5DD2	TextMode	QD-43
08A	40 5DD8	TextSize	QD-43
086	40 5E5A	TextWidth	QD-45
175	40 B6BE	TickCount	EM-22
168	40 D55A	TrackControl	CM-19
11E	40 C160	TrackGoAway	WM-26
0D0	40 9DD8	UnPackBits	
0AB	40 723C	UnionRect	QD-47
0E5	40 9136	UnionRgn	QD-57
1C1	40 DBA8	UniqueID	RM-22
1F1	40 4BBC	UnloadSeg	SL-06
1FA	40 FD32	UnlodeScrap	SM-11
00E	40 3C8A	UnmountVol	FL-17,FL-35
199	00 22A8	UpdateResFile	RM-27
054	40 4E82	UprString	OU-13
198	40 D8E2	UseResFile	RM-18
033	00 2774	VInstall	VR-06
034	00 2752	VRemove	VR-06
12A	40 C5BA	ValidRect	WM-32
129	40 C5B6	ValidRgn	WM-32
177	40 B740	WaitMouseUp	EM-20
003	40 1422	Write	FL-19,FL-41,DV-08,DV-16
038	40 4DC2	WriteParm	OU-18
1B0	40 E050	WriteResource	RM-27
0E7	40 9142	XorRgn	QD-57
1FC	40 FD96	ZeroScrap	SM-12

B

Using the Lisa Pascal Development System

Appendix B describes how the example programs in this book were developed. The development process for Macintosh programs is rapidly evolving; thus, by the time you read this, a different system will probably be in use. Chapter 3 provides a general description of the process. However, it is useful to have a specific description of how the examples were compiled and tested.

A number of steps are involved in developing an applications program for the Macintosh. Unfortunately, there are more steps than on most systems; fortunately, most steps occur automatically once the proper files are set up and used.

Editing

Currently, the files are written by an editor that is part of the Lisa Pascal workshop running on the Lisa (now the Macintosh XL).

This editor is invoked from the Lisa's main menu by typing "E". Once in the editor, you can use the Lisa mouse to move the cursor and to select items from pull-down menus, much as in typical Macintosh applications programs. If you are familiar with any of the editors on the Macintosh, it won't take you long to learn how this Lisa editor works.

Four files must be present before you can proceed: a Pascal *source* code file containing the program itself, *library* files containing external Pascal definitions and declarations, a *resource* definition file containing *resource definitions*, and an *exec* file containing a series of commands to prepare your program for running on the Macintosh. Normally, you would

write all but the library files, which are supplied by Apple as part of the Macintosh development system.

A very simple example, the "Trivial" program in Chapter 3, explains how the process works. In Chapter 3, we discussed the *resource definition file* and the *Pascal source code file*. In this appendix, we discuss the *exec* file, which runs the entire development process.

The Exec File

The exec file describes the entire preparation process. It includes all the steps for transforming the source code, library, and resource definition files of an application into a file on a Macintosh disk.

Naming the File

The name of our exec file is "clm/trivialX". Let's see how this name uses some file naming conventions recommended by Apple.

The initial "clm/" acts as a prefix and serves to uniquely identify all files written or otherwise generated in the examples developed in this book. These are the author's initials; perhaps you want to use your initials or a project name as the prefix. The prefix is part of the file name as far as the Lisa is concerned; but the Lisa sorts files alphabetically when listing its directory, so all files with the same prefix are listed together. Files supplied by Apple have several suggestive prefixes, such as "example/", "obj/", "fragment/", "intrfc/", "QD/", and "TlAsm/".

In the middle of the name, "trivial" identifies the particular example that we are developing. All files associated with this example contain this name in their full name.

The trailing "X" on our file name identifies this particular file as an exec file. Apple suggests that you place an "X" as the last character of the file name of all your exec files.

The "clm/trivialX" file has a file extension, ".text", often not explicitly mentioned. This extension identifies the file as filled with ASCII characters. Such "text" files can be edited with the Lisa editor and directly transferred to other computers through communications lines.

Using the Exec File

The exec file is run on the Lisa using the "R" command in the main menu. The syntax for running our example is:

```
<clm/trivialX
```

in response to the "Run what program?" prompt. The "<" indicates that the file is a source of input statements to the Lisa through the EXEC program on the Lisa. This left arrow symbol is followed by the file name "clm/trivialX" of our particular exec file.

Understanding the Exec File

Let's look at what our exec file does. Refer to Figure 3-2 in Chapter 3 for a diagram of the major steps in the file. These are the essential steps:

1. Call the Pascal compiler to compile your source code into intermediate code.
2. Call a *code generator* to convert this intermediate code into 68000 machine language.
3. Call a *linker* to combine it with other machine-language modules.
4. Call a *resource compiler* to combine the machine code with special data that can specify the sizes, shapes, and text in your application.
5. Call a *transfer* program to transfer your application to a Macintosh disk.
6. Call the file system to clean up by erasing the intermediate files generated during the process.
7. Call the editor to allow you to view and change the program while you test it on the Macintosh.

Here is the "clm/TrivialX" exec file.

```
$EXEC
Pclm/Trivial{compile the Pascal program}


G$M+{generate code for the Macintosh}
clm/Trivial

L{ink}clm/Trivial
obj/QuickDraw
obj/tooltraps
obj/ostraps
obj/macpaslib


clm/TrivialL
R{un}Rmaker{resource maker}
clm/TrivialR
R{un}MacCom{disk transfer program}
```

```
Fy
Lclm/Trivial.RSRC
Trivial
APPL{set type to APPL}
{set creator to ????}
N{o bundle bit}Q{uit MacCom}
F{ile system}D{elete}clm/Trivial.I
yD{elete}clm/Trivial.obj
yD{elete}clm/TrivialL.obj
yQ{uit File command}
E{dit}
$ENDEXEC
```

The first line contains the "$EXEC" command, the last line contains the "$ENDEXEC" command. These commands must bracket the list of commands contained in the exec file. Each command in that list must contain the normal keystrokes that you would type from the keyboard if you were to process the application files manually. You can insert comments within the exec file by enclosing them with curly brackets, the same as in Pascal. The "(*" brackets of Pascal, however, will not work.

Compiling

The second line, "Pclm/Trivial", starts the Pascal compiler. This line consists of a "P" command that calls the Pascal compiler from the main menu, followed by a file name that is the name of the file to be compiled. This is our Pascal *source code* file. The file name "clm/Trivial" has our own "clm/" prefix followed by the name "Trivial". The file has the file extension ".text". However, this extension is not explicitly mentioned in the exec file (or in the editor when creating the file).

Notice that there is *no* trailing letter for our source code file name, such as the "X" used for exec files. Apple recommends that you label your Pascal source code in this manner.

If you were typing these commands manually, the "P" would load the Pascal compiler, which would then prompt you for an "Input file". You would then type the name of your Pascal source code file (assuming a file extension of "text").

The next two lines of the exec file are blank. They tell the Pascal compiler that you don't want a listing file and that you want the output file from the compiler to use the same name as the source code file. The output file from the compiler is given the file extension ".I" to distinguish it from the source file.

Generating Code

The ".I" output file from the compiler contains an intermediate code that has to be processed by a *code generator* to turn it into 68000 assembly language. Other higher-level language compilers could be designed to produce such intermediate code, which could then be fed into this same code generator. Also, if Apple were to change processors, it could still use the same compiler with a different code generator.

The line "G$M+" of the exec file calls the code generator and tells it to generate code for the Macintosh rather than the Lisa.

If you typed this command manually, the "G" would load the code generator, which would prompt you for an "Input file". Instead of giving it an input file, you would type the "$M+" to inform it that it must produce Macintosh code, not Lisa code.

On the next two lines of the exec file, we specify the input file "clm/Trivial" and the output file. The output file is specified by a blank line, indicating that it has the same name as the input file. The input file assumes a file extension of ".I", making it agree with the full name of the output file from the Pascal compiler. The output file from the code generator assumes a file extension of ".OBJ".

If you typed these manually, you would type "clm/Trivial" in response to an "Input file" prompt, then hit the Return key in reponse to the "Output file" prompt.

Linking

The line "L{ink}clm/Trivial" loads the linker and tells it that its first input file is "clm/Trivial". The file extension of this input file is ".OBJ", that is, this file is the same as the file just output from the code generator. Note the comment "{ink}" following the "L". You can use this commenting technique to inform users of the full name of the command without interfering with output from the operation of the exec file.

The next few lines input the files "obj/QuickDraw", "obj/tooltraps", "obj/ostraps", and "obj/macpaslib". These input files are assumed to have the file extension "OBJ". These files contain assembly-language procedures that Macintosh applications need, including "trap" instructions to access ROM routines (see Chapter 2).

The linker starts with the first file and searches subsequent files for references to procedures and functions that are required to put the program together. It does not include procedures and functions from external files that are not needed.

The two lines following the list of input files to the linker are blank. This indicates that we have finished specifying input files and will send the listing from the linker to the Lisa console screen.

The line "clm/TrivialL" specifies the output file for the linker. The file name is the same as our source code file except for a trailing "L" to indicate that its output from the linker. It is assumed to have the file extension ".OBJ", the same file extension as the input to the linker. We could use the same name for output as for input; however, we would then have the same name for two very different files, one before the link and one after. The linker could handle this, but we might get confused if something happens during program development. Occasionally this happens, especially if you take over manual operation of the process.

If this information were entered manually, "L" would load the linker, which would then prompt us for input files until we hit a [Return]. It would ask for the listing file, then the output file, and would begin to link all input files once the output file was entered.

The Resource Maker

The next step is peculiar to the Macintosh development process. To understand it, you should know something about the structure of applications as they sit on a Macintosh disk, as described in Chapter 3.

Recall that each file in the Macintosh file system has a *data fork* and a *resource fork*. The *resource fork* contains the program code and specifications for windows, controls, and menus. In our case, the resource fork contains the contents of the "clm/TrivialL" file.

Currently, the "RMaker" program on the Lisa reads a file that you write on the Lisa called a *resource definition file* and generates a file that contains your finished application, ready to be transferred to the Macintosh.

For our "trivial" application, the *resource definition file* is "clm/TrivialR". The trailing "R" stands for "Resource". Since this is a text file written on the Lisa editor, it has the file extension ".text".

The line "R{un}Rmaker" runs the resource maker. The next line, "clm/TrivialR", tells it which resource definition file to use. As we shall see, the resource defintion file specifies the file where the code is found and the file where output from the resource maker is placed. In our case, "clm/TrivialL" contains the code, and "clm/Trivial.RSRC" is where the resulting output is placed.

Transferring the File

The next few lines cause the application to be transferred from the Lisa to a Macintosh disk. For this program, we use a transfer program called

"MacCom" that runs on the Lisa. This allows the Lisa to access files on a Macintosh disk that is placed in the Lisa's three-inch drive, the so-called lower drive.

The program "sendOne" is another way to transfer files, by sending files over a communcations line from the Lisa to the Macintosh. In this method, the Macintosh must receive the file using a program called "Disk Utility". This method requires special cabling between the Lisa and the Macintosh. It can also be hazardous: the disk utility program has a "button" that, when pressed, wipes out a Macintosh disk without asking if this is what you really want.

The "R{un}MacCom" line of the exec file loads the "MacCom" file transfer program. The next line, "Fy", tells MacCom to let you set the "Finder" information for the application. This specifies information such as the *file type*. The program "Finder" controls the system when your machine is displaying the desktop with its disk icons and file folders. In some sense, "Finder" is the primary application of the Macintosh; its job is to find and load other applications.

The line "L{isa to Mac}clm/Trivial.RSRC" tells MacCom to send the file "clm/Trivial.RSRC" to the Macintosh disk sitting in the lower drive. The next line, "Trivial", specifies what it will be called once it gets there.

The line "APPL" specifies the file type. In our case, we have set the file type equal to "APPL", which stands for application. The file type helps the finder assign the proper icon to the file and know whether it should be launched as a regular application. File types can also be used by applications programs to deal with files. For example, in Chapter 10, we use the file type "TEXT" for our text files to ensure that the example editor program accesses only files that are supposed to contain text.

The next two lines specify another kind of "Finder" information. The first is called the "creator", the second is called the "bundle" bit. In our case, we leave the "creator" equal to "????" and the "bundle" bit off. However, if you set the "creator" field equal to something more interesting (must be four ASCII characters) and you set its "bundle" bit on, then your application can attach itself to its working files by setting their "creator" fields equal to the application's "creator" field. When the system is set up in this manner, selecting one of these working files automatically loads the application first.

Also included in the bundle bit line is a "Q" to exit "macCom". The quit command also ejects your Macintosh disk, which is now ready to be placed in the Macintosh so that you can run the program.

Cleaning Up

The last few commands of the exec file erase some of the intermediate files and return to the editor. The "F" command enters the file management subsystem, and the "D" command starts the deletion process. The following files are removed from the Lisa disk: the output file "clm/Trivial.I" from the Pascal compiler, the output file "clm/Trivial.obj" from the code generator, and the output file "clm/TrivialL.obj" from the linker.

The "E" command returns you to the editor. We found this convienent because the complete development cycle goes around a number of times before the application is exact. You can now view the source code on the Lisa while the program runs on the Macintosh. With such a system, you need hardly any paper listings.

C

Disk and Volume Information

Appendix C presents an example of an application that demonstrates how to access information about volumes (disks) and files. It brings together concepts such as menus, dialogs, and alerts. We list both the Pascal source code and the complete resource definition file. This example therefore provides a model of how a complete application is put together.

This example has three menus: an Apple menu, a File menu, and an Information menu. The Apple menu gives access to an "About FileInfo" alert and the standard desk accessories (see Figure C-1). The File menu has a single entry, "Quit", which allows the user to terminate the program (see Figure C-2).

The third menu, "Information", has two entries: "volume information" and "file information" (see Figure C-3). Both cause dialogs to appear which contain information that is available without actually opening any file. Both dialogs allow some degree of interaction between the user and the Macintosh.

The "volume information" command displays a dialog with information about disks in each of the two disk drives (see Figure C-4). This information consists of the name of the volume (disk), the volume reference number, and the number of free bytes on the disk. If no disk is present in a drive, the information is blank. For each drive, there is also a button labeled "Eject" to eject the disk, and there is an "OK" button at the bottom of the display to end the dialog.

When a user clicks an "Eject" button, the corresponding disk is ejected and its information goes blank. If the user inserts a disk, then the proper information appears for the new disk. Note how the reference

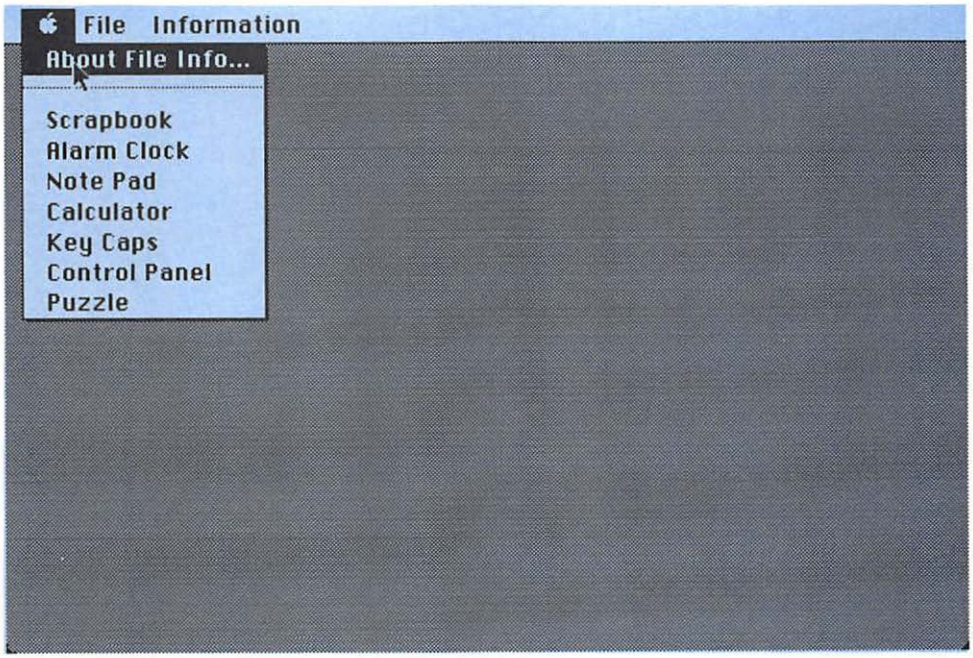
numbers work. They are always negative. The first disk that is mounted is labeled –1, the second is labeled –2, and so on. If a disk is reinserted, its original reference number is used. If the Macintosh is reset by the programmer switch or by turning it off and then on again, the sequence of reference numbers begins over with –1.

The "file information" dialog comes in two parts. First, the standard open file dialog appears (see Figure C-5). From this dialog, the user can select a particular file to be examined. The dialog for this command displays all files on the disk.

Once a file is selected, a new dialog appears that displays information about that file (see Figure C-6). It shows the file's *name*, *type*, *creator*, *folder*, *horizontal position*, *vertical position*, and *flags*. *Type* and *creator* have boxes around them, indicating that they can be edited. You can change these to any four-character combination.

The *folder* is a number that specifies the folder in which the file's icon is located. Each folder on a disk is assigned a unique integer. The main window is given the number zero. If the file's icon is on the desktop, it is given the number –2. If the file's icon is in the trash, it is given the number –3. Other folders are usually given large positive numbers.

Figure C-1. The Apple Menu



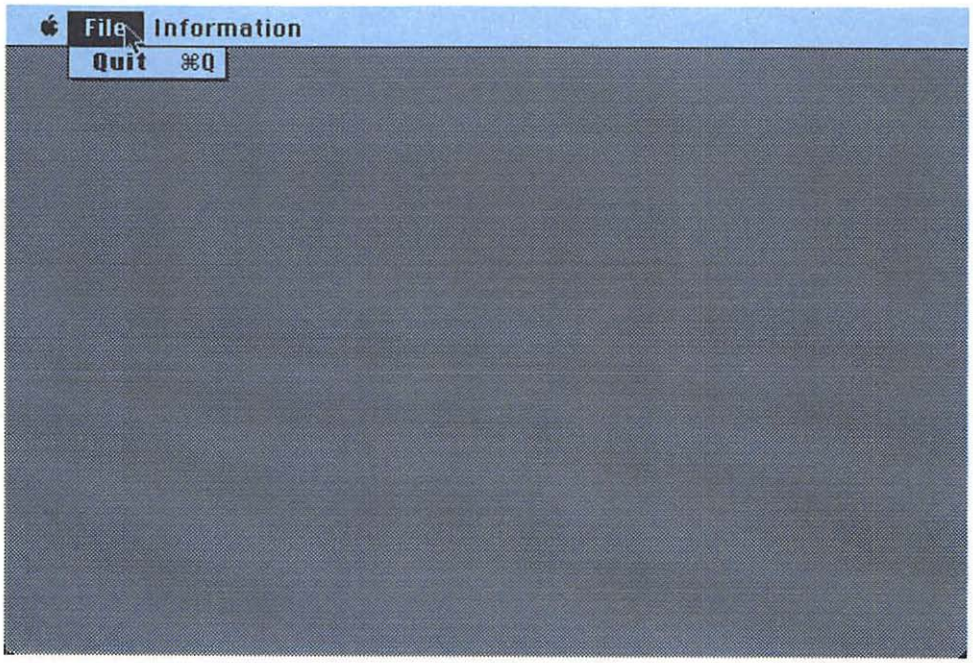
The *horizontal position* and the *vertical position* describe the position of the file's icon within its folder's window.

The *flags* field is an integer that describes certain file attributes. The values of bits in the upper byte of the flags are displayed in check boxes. The bits can be individually changed by clicking these check boxes. Some of these bits are "public", some are for internal use by the Operating System. Among the public bits are bit 5, which is the "bundle" bit; bit 6, which is the "invisible" bit; and bit 7, which is the "locked" bit.

The "bundle" bit allows an application to be involved automatically when an associated document file is selected. If an application has the bundle bit on and its "creator" type matches the "creator" type of a document, then trying to open the document causes the Finder to open this associated application first.

The "invisible" bit determines if the file's icon is visible. A value of one causes the icon to be invisible. Setting the "locked" bit prevents a file from being thrown away.

Figure C-2. The File Menu

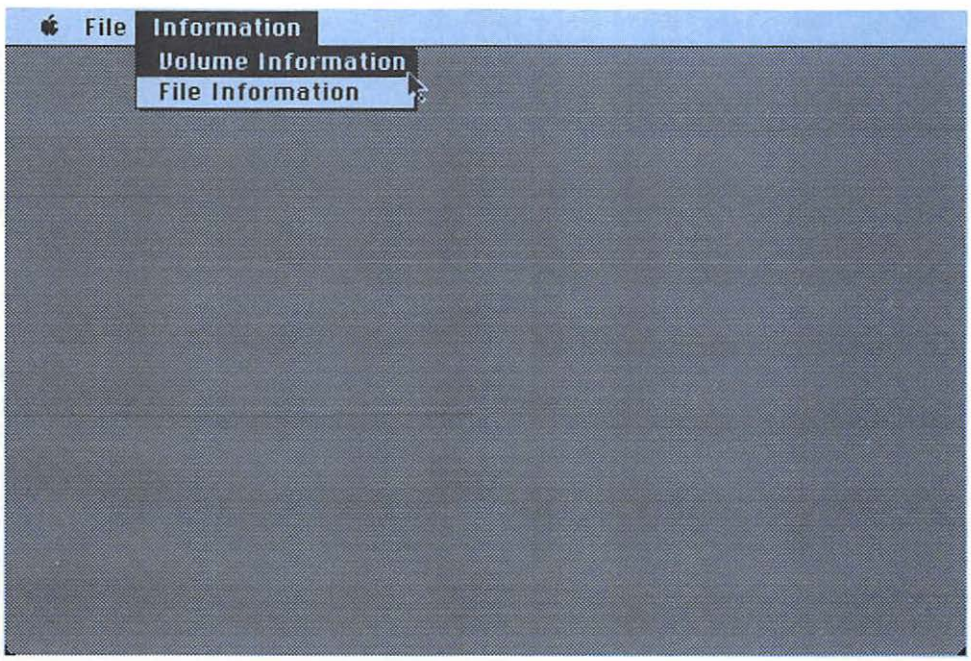


The Program

Here is the program:

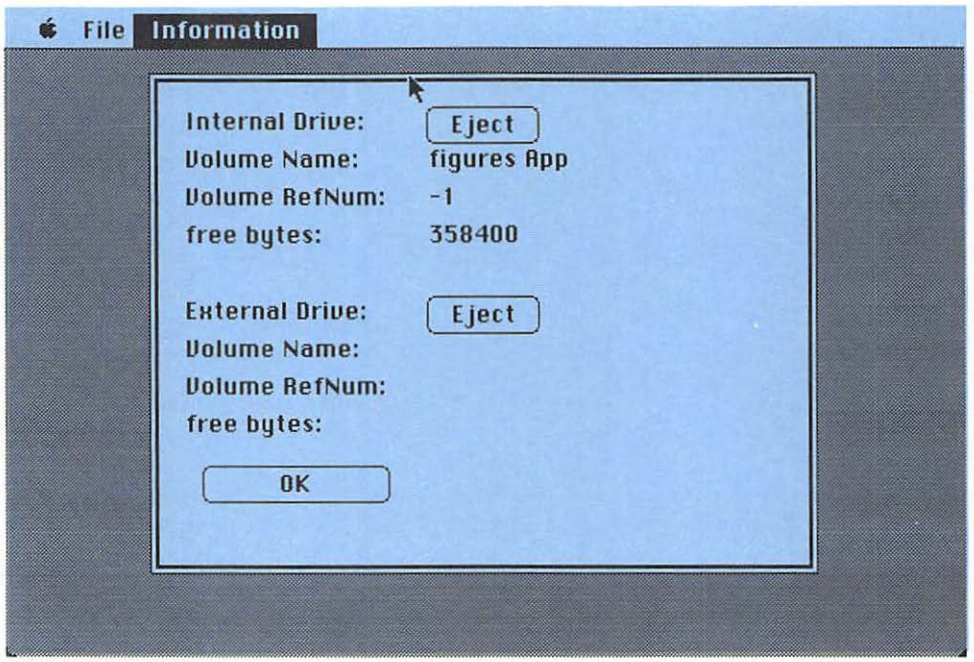
```
PROGRAM FileInfo;
   {$R-}{$X-}

   USES
      {$U obj/Memtypes    } Memtypes,
      {$U obj/QuickDraw   } QuickDraw,
      {$U obj/OSIntf      } OSIntf,
      {$U obj/ToolIntf    } ToolIntf,
      {$U Obj/PackIntf    } PackIntf;

   CONST
      {menu IDs}
      appleMenu = 1000;     { desk accessory menu}
      FileMenu  = 1001;     { File menu}
      InfoMenu  = 1002;     { Information menu}
      lastMenu  = 3;        { number of menus}
```

Figure C-3. The Information Menu



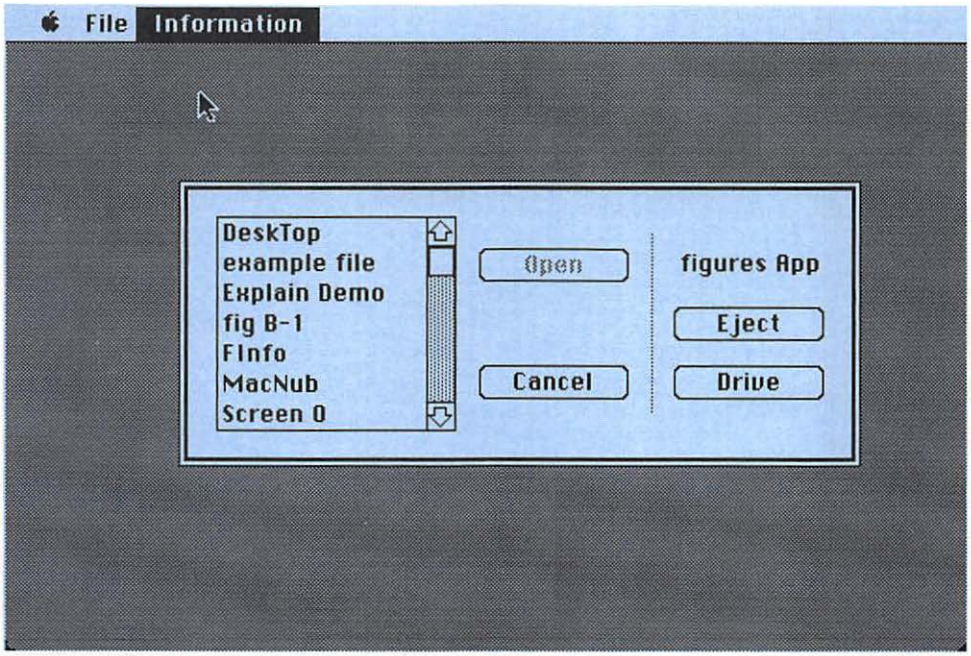
```
{common dialog and alert items}
OKBtn     = 1;
cancelBtn = 2;

{other items for volume dialog}
STV1Title = 2;
Eject1Btn = 3;
STV1TName = 4;
STV1DName = 5;
STV1TRef  = 6;
STV1DRef  = 7;
STV1Tfree = 8;
STV1Dfree = 9;
STV2Title = 10;
Eject2Btn = 11;
STV2TName = 12;
STV2DName = 13;
STV2TRef  = 14;
STV2DRef  = 15;
STV2Tfree = 16;
STV2Dfree = 17;
Insert1   = 1001;
Insert2   = 1002;
```

Figure C-4. Volume Information Dialog



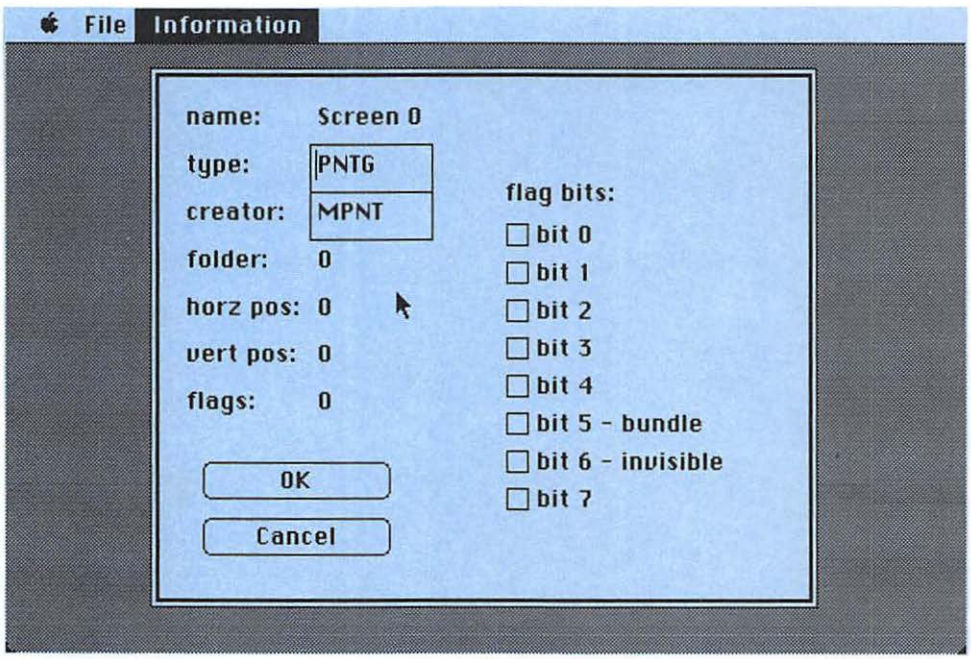
```
{other items for file dialog}
STFTName  = 3;
STFDName  = 4;
STFTType  = 5;
ETFDType  = 6;
STFTCrtr  = 7;
ETFDCrtr  = 8;
STFTFold  = 9;
STFDFold  = 10;
STFTHPos  = 11;
STFDHPos  = 12;
STFTVPos  = 13;
STFDVPos  = 14;
STFTFlag  = 15;
STFDFlag  = 16;
STFTchk   = 17;
chkBtn0   = 18;
chkBtn1   = 19;
chkBtn2   = 20;
chkBtn3   = 21;
chkBtn4   = 22;
chkBtn5   = 23;
chkBtn6   = 24;
```

Figure C-5. File Information Open Dialog



```
    chkBtn7    = 25;
    numCButtons = 8;

VAR
    done: BOOLEAN;
    where: Point;
    myMenus: ARRAY [1..lastMenu] OF MenuHandle;
    theEvt: EventRecord;
    theWindow: WindowPtr;
    theDialog: DialogPtr;
    vRefNum: INTEGER;

PROCEDURE SetLimits;
    BEGIN
        SetPt(where,100,100);
    END;

PROCEDURE SetUpMenus;
    VAR
        I : INTEGER;
    BEGIN
        InitMenus;
```

Figure C-6. The File Information Dialog



```
      myMenus[1] := GetMenu(appleMenu);
      myMenus[2] := Getmenu(FileMenu);
      myMenus[3] := Getmenu(InfoMenu);

      AddResMenu(myMenus[1],'DRVR');
      FOR I:= 1 TO lastMenu DO InsertMenu(myMenus[I],0);
      DrawMenuBar;
   END; {of SetUpMenus}

FUNCTION SetErrMess(theErr: OSErr): BOOLEAN;
   VAR
      ErrIndex, theItem: INTEGER;
      ErrMess, ErrStr: Str255;
      closeErr: BOOLEAN;
   BEGIN
      CASE theErr OF
         noErr:     ErrIndex := 2;
         bdNamErr:  ErrIndex := 4;
         fnfErr:    ErrIndex := 5;
         ioErr:     ErrIndex := 6;
         mFulErr:   ErrIndex := 7;
         nsvErr:    ErrIndex := 8;
         opWrErr:   ErrIndex := 9;
         tmfoErr:   ErrIndex := 10;
         eofErr:    ErrIndex := 11;
         paramErr:  ErrIndex := 12; {exact meaning depends}
         nsDrvErr:  ErrIndex := 13;
         dupFNErr:  ErrIndex := 14;
         dirFulErr: ErrIndex := 15;
         vLckdErr:  ErrIndex := 16;
         wPrErr:    ErrIndex := 17;
         fnOpnErr:  ErrIndex := 18;
         rfNumErr:  ErrIndex := 19;
         dskFulErr: ErrIndex := 20;
         fLckdErr:  ErrIndex := 21;
         wrPermErr: ErrIndex := 22;
         posErr:    ErrIndex := 23;
         extFSErr:  ErrIndex := 24;
         Otherwise  ErrIndex := 3;
         END;
      GetIndStr(ErrMess,1000,ErrIndex);
      NumToStr(theErr,ErrStr);
      ParamText(ErrMess,ErrStr,",");
      IF theErr <> noErr
         THEN theItem := StopAlert(1003,NIL);
      SetErrMess := (theErr <> noErr);
   END;
```

```
PROCEDURE QuitFile;
   BEGIN
      done := TRUE;
   END;

FUNCTION IHandle(theItem: INTEGER): Handle;
   VAR
      theType: INTEGER;
      ItemHdl: Handle;
      ItemBox: Rect;
   BEGIN
      GetDItem(theDialog, theItem, theType, ItemHdl, ItemBox);
      IHandle := ItemHdl;
   END;

FUNCTION CHandle(theItem: INTEGER): ControlHandle;
   BEGIN
      CHandle := ControlHandle(IHandle(theItem));
   END;

PROCEDURE SetChkBox(theItem: INTEGER);
   BEGIN
      SetCtlValue(CHandle(theItem), 1-GetCtlValue(CHandle(theItem)));
   END;

PROCEDURE ShowDNum(theNum: LongInt; theItem: INTEGER);
   VAR
      numString : Str255;
   BEGIN
      NumToString(theNum, numString);
      SetIText(IHandle(theItem), numString);
   END;

FUNCTION VFilter(theDialog: DialogPtr;
                 VAR theEvent: EventRecord;
                 VAR itemHit: INTEGER) : BOOLEAN;
   BEGIN
      ItemHit := 0;
      IF theEvent.what = nullEvent THEN
         IF GetNextEvent(diskMask, theEvent) THEN
               CASE loWord(theEvent.message) OF
                  1: ItemHit := Insert1;
                  2: ItemHit := Insert2;
                  END;
      Vfilter := (Itemhit in [Insert1, Insert2]);
   END;

PROCEDURE ShowVolInfo;
   VAR
```

```
    vRefNum, theItem: INTEGER;
    free:             LongInt;
    vName:            Str255;

  PROCEDURE ShowDr(Drive,STVDName,STVDRef,STVDFree: INTEGER);
    BEGIN
      IF GetVInfo(Drive,@vName,vRefNum,free) = noErr THEN BEGIN
        SetIText(IHandle(STVDName),vName);
        ShowDNum(vRefNum,STVDRef);
        ShowDNum(free,STVDFree);
      END;
    END;

  PROCEDURE EjectDr(Drive,STVDName,STVDRef,STVDFree: INTEGER);
    BEGIN
      IF Eject(NIL,Drive) = noErr THEN BEGIN
        SetIText(IHandle(STVDName),");
        SetIText(IHandle(STVDRef),");
        SetIText(IHandle(STVDFree),");
      END;
    END;

  BEGIN
    theDialog := GetNewDialog(1001,NIL,POINTER(-1));
    ShowDr(1,STV1DName,STV1DRef,STV1DFree);
    ShowDr(2,STV2DName,STV2DRef,STV2DFree);

    FlushEvents(everyEvent,0);
    REPEAT
      ModalDialog(@VFilter,theItem);
      CASE theItem OF
        Insert1:   ShowDr (1,STV1DName,STV1DRef,STV1DFree);
        Insert2:   ShowDr (2,STV2DName,STV2DRef,STV2DFree);
        Eject1Btn: EjectDr(1,STV1DName,STV1DRef,STV1DFree);
        Eject2Btn: EjectDr(2,STV2DName,STV2DRef,STV2DFree);
        END;
    UNTIL theItem = OKBtn;
    DisposDialog(theDialog);
  END;

PROCEDURE ShowFileInfo;
  TYPE
    BtnArray = PACKED ARRAY [0..15] OF BOOLEAN;

  VAR
    I, ItemHit:     INTEGER;
    typeList:       SFTypeList;
    reply:          SFReply;
    theFInfo:       FInfo;
```

```
   fType, fCreator: Str255;
   CArray:          BtnArray;

PROCEDURE FLCall(theErr: OSErr);
   BEGIN
      IF SetErrMess(theErr) THEN Exit(ShowFileInfo);
   END;

BEGIN
   REPEAT
      SFGetFile(where,'',NIL,-1,typeList,NIL,reply);
      IF reply.good THEN BEGIN
         FLCall(GetFInfo(reply.fName, reply.vRefNum, theFInfo));
         theDialog := GetNewDialog(1002,NIL,POINTER(-1));

         CArray := BtnArray(theFInfo.fdFlags);
         FOR I := 0 TO numCButtons-1 DO
            SetCtlValue(CHandle(chkBtn0+I),ORD(CArray[I]));

         SetIText(IHandle(STFDName),reply.fName);

         fType := '    ';
         FOR I := 1 TO 4 DO fType[I] := theFInfo.fdType[I];
         SetIText(IHandle(ETFDType),fType);

         fCreator := '    ';
         FOR I := 1 TO 4 DO fCreator[I] := theFInfo.fdCreator[I];
         SetIText(IHandle(ETFDCrtr),fCreator);

         ShowDNum(theFInfo.fdFldr,STFDFold);
         ShowDNum(theFInfo.fdFlags,STFDFlag);
         ShowDNum(theFInfo.fdLocation.h,STFDHPos);
         ShowDNum(theFInfo.fdLocation.v,STFDVPos);

         FlushEvents(everyEvent,0);
         REPEAT
            ModalDialog(NIL,itemHit);
            IF itemHit in [chkBtn0..ChkBtn7] THEN BEGIN
               SetChkBox(itemHit);
               FOR I := 0 TO numCButtons-1 DO
                  CArray[I] := GetCtlValue(CHandle(chkBtn0+I))<>0;
               ShowDNum(INTEGER(CArray),STFDFlag);
            END;
         UNTIL itemHit in [OKBtn,CancelBtn];

         IF itemHit = OKBtn THEN BEGIN
            theFInfo.fdFlags := INTEGER(CArray);
```

```
                GetIText(IHandle(ETFDType),fType);
                FOR I := 1 TO 4 DO
                    theFInfo.fdType[I] := fType[I];

                GetIText(IHandle(ETFDCrtr),fCreator);
                FOR I := 1 TO 4 DO
                    theFInfo.fdCreator[I] := fCreator[I];

                FLCall(SetFInfo(reply.fName,reply.vRefNum,theFInfo));
                END;
            DisposDialog(theDialog);
        END;
    UNTIL NOT reply.good;
END;

PROCEDURE SetUpSys;
BEGIN
    InitGraf(@thePort);
    InitFonts;
    InitWindows;
    TEInit;
    InitDialogs(NIL);
    SetEventMask(everyEvent);
    FlushEvents(everyEvent,0);

    SetLimits;
    SetUpMenus;
    InitCursor;
    vRefNum := 0;
    done := FALSE;
END;

PROCEDURE UpdateSys;
BEGIN
    SystemTask;
END;

PROCEDURE DoAppleMenu(theItem: INTEGER);
VAR
    refNum: INTEGER;
    name:   Str255;
BEGIN
    If theItem = 1
        THEN theItem := Alert(1004,NIL)
        ELSE
            BEGIN
                GetItem(myMenus[1],theItem,name);
                refNum := OpenDeskAcc(name);
            END;
END;
```

```
PROCEDURE DoFileMenu(theItem: INTEGER);
   BEGIN
      CASE theItem OF
         1: QuitFile;
      END;
   END;

PROCEDURE DoInfoMenu(theItem: INTEGER);
   BEGIN
      CASE theItem OF
         1: ShowVolInfo;
         2: ShowFileInfo;
      END;
   END;

PROCEDURE SelectMenu(selection : LongInt);
   BEGIN
      CASE HiWord(selection) OF
         appleMenu:   DoAppleMenu(LoWord(selection));
         FileMenu:    DoFileMenu (LoWord(selection));
         InfoMenu:    DoInfoMenu (LoWord(selection));
         END;
      HiliteMenu(0); {to unhighlight selected menu in menu bar}
   END;

PROCEDURE KeyEvent(theKey: Char);
   BEGIN
      IF BitTst(@theEvt.modifiers,7) {check for command key}
         THEN SelectMenu(MenuKey(theKey));
   END; {KeyEvent}

PROCEDURE WindowUpdate;
   BEGIN
      theWindow := windowPtr(theEvt.message);
      SetPort(theWindow);
      BeginUpdate(theWindow);
         EraseRect(theWindow^.portRect);
         DrawControls(theWindow);
         DrawGrowIcon(theWindow);
      EndUpdate(theWindow);
   END; {Update}

PROCEDURE WindowActivate;
   BEGIN
      WindowUpdate;
   END; {Activate}

BEGIN  {main program}
   SetUpSys;
```

```
    REPEAT
       UpdateSys;
       IF GetNextEvent(everyEvent,theEvt) THEN
          CASE theEvt.what OF
             mouseDown:
                CASE FindWindow(theEvt.where,theWindow) OF
                   inMenuBar:    SelectMenu(MenuSelect(theEvt.where));
                   inSysWindow:  SystemClick(theEvt,theWindow);
                   END;
              keyDown, autoKey:  KeyEvent(Chr(theEvt.message MOD 256));
              updateEvt:         WindowUpdate;
              activateEvt:       WindowActivate;
              END; {of what event}
    UNTIL done;
END.
```

The Resource Definition File

Here is its complete resource definition file:

```
*    Resource Definition File for FileInfo Demo

clm/FInfo.Rsrc

Type MENU
  ,1000
  \14
       About File Info...
       (----------------

  ,1001
  File
     Quit/Q <B

  ,1002
  Information
     Volume Information
     File Information


Type STR#
  ,1000
Untitled
No error
Unknown Error
Bad file or volume Name
File not found
Disk I/O Error
```

```
Memory full
No such volume
File already open for writing
Too many files open
End of file
Bad number
No such drive
Duplicate file name
Directory full
Software volume lock
Hardware volume lock
File not open
Bad reference number
Disk full
Permission denied to access file
File position out of range
External file system


Type DLOG
  ,1001
  40 85 290 425
  Visible 1 NoGoAway 0
  1001
Volume Display Window


Type DITL
  ,1001
  17

  BtnItem Enabled
  200 20 220 120
OK

  StatText Disabled
  10 10 30 140
Internal Drive:

  BtnItem Enabled
  10 140 30 200
Eject

  StatText Disabled
  30 10 50 140
Volume Name:

  StatText Disabled
  30 140 50 320
```

```
  StatText Disabled
  50 10 70 140
Volume RefNum:

  StatText Disabled
  50 140 70 320


  StatText Disabled
  70 10 90 140
free bytes:

  StatText Disabled
  70 140 90 320


  StatText Disabled
  110 10 130 140
External Drive:

  BtnItem Enabled
  110 140 130 200
Eject

  StatText Disabled
  130 10 150 140
Volume Name:

  StatText Disabled
  130 140 150 320


  StatText Disabled
  150 10 170 140
Volume RefNum:

  StatText Disabled
  150 140 170 320


  StatText Disabled
  170 10 190 140
free bytes:

  StatText Disabled
  170 140 190 320
```

```
Type DLOG
  ,1002
  40 85 310 425
  Visible 1 NoGoAway 0
  1002
File Display Window


Type DITL
  ,1002
  25

  BtnItem Enabled
  200 20 220 120
OK

  BtnItem Enabled
  230 20 250 120
Cancel

  StatText Disabled
  10 10 30 70
name:

  StatText Disabled
  10 80 30 320


  StatText Disabled
  35 10 55 70
type:

  EditText Disabled
  35 80 55 140


  StatText Disabled
  60 10 80 70
creator:

  EditText Disabled
  60 80 80 140


  StatText Disabled
  85 10 105 80
folder:
```

```
  StatText Disabled
  85 80 105 140


  StatText Disabled
  110 10 130 80
horz pos:

  StatText Disabled
  110 80 130 140


  StatText Disabled
  135 10 155 80
vert pos:

  StatText Disabled
  135 80 155 140


  StatText Disabled
  160 10 180 80
flags:

  StatText Disabled
  160 80 180 140


  StatText Disabled
  50 180 70 250
flag bits:

  ChkItem Enabled
  70 180 90 250
bit 0

  ChkItem Enabled
  90 180 110 250
bit 1

  ChkItem Enabled
  110 180 130 250
bit 2

  ChkItem Enabled
  130 180 150 250
bit 3
```

```
  ChkItem Enabled
  150 180 170 250
bit 4

  ChkItem Enabled
  170 180 190 320
bit 5 - bundle

  ChkItem Enabled
  190 180 210 320
bit 6 - invisible

  ChkItem Enabled
  210 180 230 250
bit 7

Type ALRT
  ,1003
  100 70 200 440
  1003
  7654

Type DITL
  ,1003
  3

  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 150 50 360
File Error: ^0

  StatText Disabled
  60 150 90 360
ID number: ^1

Type ALRT
  ,1004
  100 70 200 450
  1004
  4444

Type DITL
  ,1004
  3
```

```
  BtnItem Enabled
  70 10 90 100
OK

  StatText Disabled
  10 10 30 370
FileInfo, a demonstration program for files

  StatText Disabled
  30 10 50 360
Christopher L. Morgan, 1985


Type CODE
  clm/FInfoL,0
```

Descriptions

This appendix describes only the key routines; in this case, the procedures that implement the file information dialog.

First, we discuss a filter routine for one of the information dialogs, then the procedures to implement both of the file information menu items.

A Modal Dialog Filter

The function "VFilter" serves as a filter procedure for the modal dialog in the "ShowVolInfo" procedure. It modifies events generated in the dialog before they are sent to the user as item "hits". This particular filter modifies null events, turning one of them into a special disk insertion event whenever a disk is inserted into a disk drive.

A dialog filter procedure has three parameters: a dialog pointer that points to the current dialog, an event record passed by reference, and an integer passed by reference that contains the code of the item that was hit. A dialog filter function returns a value of false if the event from the filter needs further processing by the dialog, true if we want the modal dialog to return immediately with the results of our filter.

The procedure begins by setting "ItemHit" equal to zero. This initializes it to a neutral value for a test at the end of the routine. If the ".what" field of the event record indicates the null event, then we call "GetNextEvent" with an event mask equal to "diskMask" to pick up any disk insertion events. We then check the low word of the ".message" field of the event record to determine in which drive the disk was inserted. If it is inserted in the internal drive (drive 1), we set "ItemHit" equal to the constant "Insert1"; if it is inserted into the external drive (drive 2), we

set "ItemHit" equal to the constant "Insert2". These constants are larger than any possible item numbers from this dialog. If we set "ItemHit" equal to one of these values, then we set the Boolean return value of our filter function equal to true, causing the modal dialog to immediately return with one of these item numbers in "ItemHit".

Displaying Volume Information

The procedure"ShowVolInfo" implements the "volume information" command of the Information menu. It has four local parameters: the integers "vRefNum" and "theItem" to hold a volume reference number and the item number, the long integer "free" to contain the number of free bytes on a disk, and a string to hold the volume name.

The "ShowVolInfo" procedure has two subprocedures: "ShowDr" displays the volume information for a selected drive, and "EjectDr" ejects a disk from a specified disk drive and erases the volume information.

Let's look at the "ShowDr" and "EjectDr" routines. Both routines have four integer parameters: "Drive" specifies the drive, "STVDName" specifies the item number of the static text item where the volume name is displayed, "STVDRef" specifies the item number of the static text item where the volume reference number is displayed, and "STVDFree" specifies the item number of the static text item where the number of free bytes on the disk is displayed.

The "ShowDr" procedure calls the File Manager's "GetVInfo" routine to get the volume information. It returns the volume name, volume reference number, and number of free bytes on the disk. If this routine is successful, we call "SetIText" to place the volume name in the dialog and "ShowDNum" to display the volume reference number and number of free bytes on the disk.

The "EjectDr" procedure calls the File Manager's "Eject" routine to eject the disk from the specified drive. If this is successful, the procedure calls "SetIText" three times to place blank strings in the appropriate static text items of the dialog.

The "ShowVolInfo" procedure is like a small version of the entire program. It has an initialization section and an event loop implemented with a REPEAT..UNTIL loop. The initialization section begins by calling "GetNewDialog" to grab the dialog definition from the resource file, display the dialog, and return a handle to it. The resource definition of this dialog and its dialog list can be found in the resource definition file as DLOG and DITL resources, both with resource ID number 1001. We store this handle in "theDialog".

The procedure continues by calling "ShowDr" twice to attempt to show the volume information for both drives. If no disk is in a drive or if the drive isn't attached, no information is displayed. Finally, it calls "FlushEvents" to remove all events before going into the event loop.

The event loop calls "ModalDialog" to get the item number of the item hit. In this program, we pass a pointer to our filter as the first parameter of "ModalDialog" and pass "theItem" as the second. In this way, "theItem" gets the item number. A CASE statement detects which item is hit and acts accordingly. If it is the "Insert1" item generated by our filter routine when a disk was inserted into the internal drive, then we call our "ShowDr" to show the volume information for the disk in the internal drive. Likewise, in response to a value of "Insert2", we show the information for the external drive. A value of "Eject1Btn" means that the first eject button was selected. In response, we call "EjectDr" to try to eject the disk (if any) in the internal drive. A value of "Insert2" means that we should call "EjectDr" to eject the disk in the external drive. The event loop continues until the OK button is detected.

Before the procedure ends, it calls "DisposDialog" to dispose of the dialog, wiping it from view.

Displaying File Information

The procedure "ShowFileInfo" implements the "File Information" command of the Information menu, displaying a dialog filled with file information, some of which can be changed.

It has one local type and a number of local variables. The local type is called "BtnArray" and is defined as a packed array of 16 Boolean variables. Each Boolean is stored as a bit.

The first two local variables are integers: "I" is used as an index to a FOR loop, and "ItemHit" holds an item number from the file dialog. The next two local variables are used with the standard file get routine. Here, "typelist" is of type "SFtypelist" and "reply" is of type "SFReply". We use these types to open and save files. The next local variable, "theFInfo", is of type "FInfo" and is a record structure that holds file information. It is defined as follows:

```
FInfo = RECORD
          fdType:       OSType;
          fdCreator:    OSType;
          fdFlags:      INTEGER;
          fdLocation:   Point;
          fdFldr:       INTEGER;
        END;
```

These are the fields that we wish to display. The first field, ".fdType", is a four-character file type. The second field, ".fdCreator", is a four-character designator to help a document locate its associated applications program. The next, ".fdFlags", holds certain file attributes, such as "locked", "bundle", and "invisible". The next field, ".fdLocation", is a point that gives the location of the file's icon within its window. Finally, the field ".fdFldr" is an integer that identifies which folder contains the file's icon.

The next two local variables, "fType" and "fCreator", are strings to hold the file type and creator fields. The last local variable, "CArray", is of type "BtnArray", which we defined above. We use this variable to hold the bits of the "fdFlags" field.

The "ShowFileInfo" procedure has one subprocedure, "FLCall", which handles errors.

The "ShowFileInfo" procedure consists of an outer REPEAT..UNTIL loop that first calls the Standard File Package's "SFGetFile" procedure used in the "OpenFile" procedure. Here, we instruct it to display files of all types by passing –1 in the fourth parameter. The last parameter is the "reply", which contains fields that specify if a valid file has been selected. If a valid file is selected, this parameter gives the file name and volume reference number.

If there is a valid selection ("reply.good" is true), then we call "GetFInfo" to get the file information. Notice that we don't have to open the file to make this call. The "GetFInfo" routine expects three parameters: the file name, the volume reference number, and a variable of type "fInfo" passed by reference.

We next call "GetNewDialog" to set up the file dialog. It gets the dialog's definition from the resource file and displays the dialog on the screen. We store the resulting handle in the variable "theDialog". The resource definition for this dialog and its dialog list can be found in the resource definition file as resources of type DLOG and DITL, both with ID 1002.

Next, we copy the ".fdFlags" field into our button array, coercing the type from an integer to a packed array of 16 Booleans. We then load the first eight of these into the control values for our eight check boxes. We use the ORD function to convert each Boolean into a long integer for the control value.

We call "SetIText" to put the file name into the corresponding dialog item. We move the file type and creator field into strings, one character at a time, and display the resulting strings in the dialog. These are placed in editable text items because they can be edited by the user.

We use our "ShowDNum" to display the ".fdFldr", ".fdFlags", and both coordinates of the file's location as numbers in the dialog.

Next comes a REPEAT..UNTIL loop, which acts like an event loop for our file dialog. It calls "ModalDialog" to see which item is selected. If it is a check box, we call our "SetChkBox" to click the check box, then update the numerical value displayed for the flags. The REPEAT loop continues until the user hits the OK button or cancel button.

After the REPEAT loop, we update the ".fdFlags", ".fdType", and ".fdCreator" fields, then call the File Manager's "SetFInfo" routine to place this new file information back onto the disk. Finally, we dispose of the dialog.

This concludes our discussion of the "FileInfo" example. You are welcome to type it in and give it a try.

D

Macintosh Routines Used in Example Programs

Appendix D contains a table of the built-in Macintosh routines used in this book. They were sorted alphabetically by the manager.

Control Manager
CM-DrawControls
CM-FindControl
CM-GetCRefCon
CM-GetCtlValue
CM-GetNewControl
CM-HideControl
CM-HiliteControl
CM-MoveControl
CM-SetCRefCon
CM-SetCtlMax
CM-SetCtlValue
CM-ShowControl
CM-SizeControl
CM-TrackControl

Dialog Manager
DL-DisposDialog
DL-DlgCopy [Pascal only]
DL-DlgCut [Pascal only]
DL-DlgPaste [Pascal only]
DL-GetDItem
DL-GetItext
DL-GetNewDialog
DL-InitDialogs
DL-ModalDialog
DL-NoteAlert
DL-ParamText
DL-SelIText
DL-SetDAFont [Pascal only]
DL-SetIText
DL-StopAlert

Desk Manager
DS-OpenDeskAcc
DS-SystemClick
DS-SystemEdit
DS-SystemTask

Event Manager
EM-Button
EM-FlushEvents
EM-GetKeys
EM-GetMouse
EM-GetNextEvent
EM-SetEventMask

File Manager
FL-Create
FL-Eject
FL-FSClose
FL-FSOpen
FL-FSRead
FL-FSWrite

FL-GetEOF
FL-GetFInfo
FL-GetVInfo
FL-SetFInfo

Font Manager
FM-InitFonts

Memory Manager
MM-HLock
MM-HUnLock
MM-SetHandleSize

Menu Manager
MN-AddResMenu
MN-CheckItem
MN-DisableItem
MN-DrawMenuBar
MN-EnableItem
MN-GetItem
MN-GetMenu
MN-HiliteMenu
MN-InitMenus
MN-InsertMenu
MN-MenuKey
MN-MenuSelect
MN-NewMenu

Operating System Utilities
OU-SysBeep

Package Manager
PK-GetIndStr
PK-NumToString
PK-SFGetFile
PK-SFPutFile

QuickDraw
QD-ClipRect
QD-ClosePicture
QD-ClosePoly
QD-CloseRgn
QD-DrawChar
QD-DrawPicture
QD-DrawString
QD-EraseOval
QD-EraseRect
QD-EraseRgn
QD-EraseRoundRect
QD-FillOval
QD-FillRect
QD-FillRgn
QD-FillRoundRect
QD-FrameOval
QD-FramePoly
QD-FrameRect
QD-FrameRgn
QD-FrameRoundRect
QD-GetPort
QD-GlobaltoLocal
QD-HideCursor
QD-InitCursor
QD-InitGraf
QD-InverRoundRect
QD-InvertOval
QD-InvertRect
QD-Line
QD-LineTo
QD-Move
QD-MovePortTo
QD-MoveTo
QD-NewRgn
QD-OffsetPoly
QD-OffsetRgn
QD-OpenPicture
QD-OpenPoly
QD-OpenPort
QD-OpenRgn
QD-PaintOval
QD-PaintRect
QD-PaintRoundRect
QD-PenMode

QD-PenNormal
QD-PenPat
QD-PenSize
QD-PenSize
QD-PortSize
QD-PtInRgn
QD-RectInRgn
QD-ScrollRect
QD-SetCursor
QD-SetOrigin
QD-SetPort
QD-SetPt
QD-SetRect
QD-SetRectRgn
QD-StuffHex
QD-TextFace
QD-TextFont
QD-TextSize
QD-UnionRgn

Text Edit
TE-TEActivate
TE-TECalText
TE-TEClick
TE-TECopy
TE-TECut
TE-TEDeactivate
TE-TEDispose
TE-TEIdle
TE-TEInit
TE-TEKey
TE-TENew
TE-TEPaste
TE-TEScroll
TE-TEUpdate

Toolbox Utilities
TU-BitSet
TU-BitTst
TU-HiWord
TU-LoWord

Window Manager
WM-BeginUpdate
WM-DragWindow
WM-DrawGrowIcon
WM-EndUpdate
WM-FindWindow
WM-FrontWindow
WM-GetNewWindow
WM-GetWRefCon
WM-GrowWindow
WM-HideWindow
WM-InitWindows
WM-InvalRect
WM-SelectWindow
WM-SetWTitle
WM-ShowWindow
WM-SizeWindow
WM-TrackGoAway

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

Look for these other Plume/Waite titles on the Macintosh®:

☐ **Games and Utilities for the Macintosh® by Dan Shafer.** Thirty exciting games and useful utility programs in Macintosh Pascal, ready for you to type in and run. Something for everyone, from "Crypto-quotes," "Parachute Man," and "Logic Probe," to sort routines and icon and menu constructors. Full-sized and expertly written, these programs are not only entertaining and useful, they are also a valuable education in the finer points of Macintosh programming. (256410—$18.95)

☐ **Pascal Primer for the Macintosh® by Dan Shafer.** A friendly, easy-to-follow introduction to Apple's exciting new version of Pascal. For first-time programmers as well as those familiar with earlier, less sophisticated versions of this important language. Extensive hands-on examples, exercises, and a relaxed, supportive style make learning Macintosh Pascal easy, even for the novice. Covers files, events, QuickDraw, windows, the mouse, and more. (256402—$19.95)

☐ **Basic Primer for the Macintosh® by Emil Flock and Miriam Flock.** Apple's own Macintosh Basic is one of the best-structured, fastest, easiest-to-learn versions of Basic ever developed. Using entertaining, carefully graded programming examples, this book takes the complete novice from simple one-line programs to full mastery of the language. Later chapters cover such advanced topics as sound, files, and using the Mac's QuickDraw and Toolbox routines.

☐ **Assembly Language Primer for the Macintosh® by Keith Mathews.** Many serious application programs must be written in assembly language, which alone has the speed and versatility to handle tough problems. Assuming no previous knowledge of assembly language, this books shows you, in easy, step-by-step style, how to master 68000 code, and at the same time, how to access all of the Mac's features from your programs: windows, the mouse, text editing, and more.

Plume/Waite books on the TRS-80® Model 100:

☐ **Introducing the TRS-80® Model 100, by Diane Burns and S. Venit.** This book, intended for newcomers to the Model 100, offers simple step-by-step explanations of how to set up your Model 100 and how to use its built-in programs: TEXT, ADDRSS, SCHEDL, TELCOM, and BASIC. Specific instructions are given for connecting the Model 100 to the cassette recorder, other computers, the telephone lines, the optional disk drive/video interface, and the optional bar code reader. (255740—$15.95)

☐ **Mastering BASIC on the TRS-80® Model 100, by Bernd Enders.** An exceptionally easy-to-follow introduction to the built-in programming language on the Model 100. Also serves as a comprehensive reference guide for the advanced user. Covers all Model 100 BASIC features including graphics, sound, and file-handling. With this book and the Model 100 you can learn BASIC anywhere! (255759—$19.95)

☐ **Games and Utilities for the TRS-80® Model 100, by Ron Karr, Steven Olsen, and Robert Lafore.** A collection of powerful programs to enhance your Model 100. Enjoy fast-paced, exciting card games, arcade games, music, art, and learning games. Help yourself to practical utilities that let you count words in a text file, turn your Model 100 into a scientific calculator, show file sizes, and generally increase your Model 100's usefulness, and your own grasp of programming. (255775—$16.95)

☐ **Practical Finance on the TRS-80® Model 100, by S. Venit and Diane Burns.** The perfect book for anyone using the Model 100 in business: investors, real estate brokers, managers. Contains short but powerful programs to perform production planning, and access financial and other information from CompuServe® and the Dow Jones News/Retrieval® service. (255767—$15.95)

☐ **Hidden Powers of the TRS-80® Model 100, by Christopher L. Morgan.** This amazing book takes you deep inside the Model 100 to reveal for the first time how it really works. You'll learn about the amazing power buried in the ROM, and how to use this power in your own programs. You can print in reverse video, prevent any screen lines from scrolling, dial the telephone from BASIC, control external devices from the cassette port, and discover many other fascinating secrets hidden within your Model 100. (255783—$19.95)

Other Plume/Waite books available from New American Library:

☐ **BASIC PRIMER for the IBM® PC and XT by Bernd Enders and Bob Petersen.** An exceptionally easy-to-follow entry into BASIC programming that also serves as a comprehensive reference guide for the advanced user. Includes thorough coverage of all IBM BASIC features: color graphics, sound, disk access, and floating point. (254957—$16.95)

☐ **DOS PRIMER for the IBM® PC and XT by Mitchell Waite, John Angermeyer and Mark Noble.** An easy-to-understand guide to IBM's disk operating system, versions 1.1 and 2.0, which explains—from the ground up—what a DOS does and how to use it. Also covered are advanced topics such as the fixed disk, tree-structured directories, and redirection. (254949—$14.95)

☐ **PASCAL PRIMER for the IBM® PC by Michael Pardee.** An authoritative guide to this important structured language. Using sound and graphics examples, this book takes the reader from simple concepts to advanced topics such as files, linked lists, compilands, pointers, and the heap. (254965—$17.95)

☐ **ASSEMBLY LANGUAGE PRIMER for the IBM® PC and XT by Robert Lafore.** This unusual book teaches assembly language to the beginner. The author's unique approach, using DEBUG and DOS functions, gets the reader programming fast without the usual confusion and overhead found in most books on this fundamental subject. Covers sound, graphics, and disk access. (254973—$24.95)

☐ **BLUEBOOK OF ASSEMBLY ROUTINES for the IBM® PC and XT by Christopher Morgan.** A collection of expertly written "cookbook" routines that can be plugged in and used in any BASIC, Pascal, or assembly language program. Included are graphics, sound, arithmetic conversions. Get the speed and power of assembly language in your program, even if you don't know the language! (254981—$19.95)

COMPUTER • Z5643 • $24.9
CANADA • $33.2

The Waite Group

HIDDEN POWERS OF THE MACINTOSH®

Over 400 built-in ROM routines give the Macintosh® a power no personal computer has had before. But to program the Mac you must know the concepts behind these routines and how to harness them.

Using simple working program examples, this book follows a logical, step-by-step approach to explain how to access the Mac's built-in software. You'll learn how these routines are grouped into managers, like the event manager, menu manager, and window manager; about such graphics concepts as GrafPorts, bit-maps, regions, and clipping; and about events, files, and memory management. Special attention is paid to fundamental programming concepts like handles and pointers as they are used on the Mac.

Apple's own Development Pascal is used as a model, but all discussions are general enough that the book is equally applicable to software development in any language. The program examples not only demonstrate the concepts involved but can also be used to generate full-blown applications programs.

The Waite Group is a Sausalito, California-based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty-five titles, including such best sellers as *Pascal Primer for the Macintosh,®* *Assembly Language Primer for the IBM PC & XT,* *Bluebook of Assembly Routines for the IBM PC & XT,* and *DOS Primer for the IBM PC & XT.* Internationally known and award winning, Waite Group books are distributed worldwide, and have been repackaged with the products of such major companies as Epson, Wang, Xerox, Tandy Radio-Shack, NCR and Exxon. Mr. Waite, President of the Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.

ISBN 0-452-25643-7